Processing Mesh Animations

From Static to Dynamic Geometry and Back

Doctoral Dissertation submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Tim Winkler

under the supervision of
Prof. Kai Hormann

February 2011

Dissertation Committee

Prof. Antonio Carzaniga	Università della Svizzera Italiana, Switzerland
Prof. Kai Hormann	Università della Svizzera Italiana, Switzerland
Prof. Fabian Kuhn	Università della Svizzera Italiana, Switzerland
Prof. Mario Botsch	University of Bielefeld, Germany
Prof. Craig Gotsman	Technion Haifa, Israel

Dissertation accepted on 14 February 2011

Prof. Kai Hormann
Research Advisor
Università della Svizzera Italiana, Switzerland

Prof. Michele Lanza
PhD Program Director

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Tim Winkler
Lugano, 14 February 2011

To my family

"Some things in life are bad
They can really make you mad
Other things just make you swear
and curse.
When you're chewing on life's gristle
Don't grumble, give a whistle
And this'll help things turn out for
the best..."

–Monty Python

Abstract

Static triangle meshes are the representation of choice for artificial objects, as well as for digital replicas of real objects. They have proven themselves to be a solid foundation for further processing. Although triangle meshes are handy in general, it may seem that their *discrete* approximation of reality is a downside. But in fact, the opposite is true. The approximation of the real object's shape remains the same, even if we willfully change the vertex positions in the mesh, which allows us to *optimize* it in this way. Due to modern acquisition methods, such a step is always beneficial, often even required, prior to further processing of the acquired triangle mesh.

Therefore, we present a general framework for optimizing surface meshes with respect to various target criteria. Because of the simplicity and efficiency of the setup it can be adapted to a variety of applications. Although this framework was initially designed for single static meshes, the application to a set of meshes is straightforward. For example, we convert a set of meshes into compatible ones and use them as basis for creating dynamic geometry.

Consequently, we propose an *interpolation* method which is able to produce visually plausible interpolation results, even if the compatible input meshes differ by large rotations. The method can be applied to any number of input vertex configurations and due to the utilization of a hierarchical scheme, the approach is fast and can be used for very large meshes.

Furthermore, we consider the opposite direction. Given an animation sequence, we propose a pre-processing algorithm that considerably reduces the number of meshes required to describe the sequence, thus yielding a *compact representation*. Our method is based on a clustering and classification approach, which can be utilized to automatically find the most prominent meshes of the sequence. The original meshes can then be expressed as linear combinations of these few representative meshes with only small approximation errors.

Finally, we investigate the *shape space* spanned by those few meshes and show how to apply different interpolation schemes to create other shape spaces, which are not based on vertex coordinates. We conclude with a careful analysis of these shape spaces and their usability for a compact representation of an animation sequence.

Acknowledgements

First and foremost, I would like to express my sincere appreciation for the support provided by my advisor Prof. Kai Hormann. I am very grateful for his constant advice, his constructive criticism, and mostly his patient encouragement and sincerity. Each time I had climbed one notch, he still had more notches to offer, thus fostering constant progress on my way of pursuing a Ph.D.

Undoubtedly, I would also like to thank both external reviewers Prof. Mario Botsch from the University of Bielefeld and Prof. Craig Gotsman from the Technion in Haifa, as well as both internal committee members Prof. Antonio Carzaniga and Prof. Fabian Kuhn for reading and evaluating this thesis.

Since I spent my postgraduate life mainly at two different universities, I would further like to thank my former colleagues at the Clausthal University of Technology who either supported me in the first years or from time to time cheered me up when it was necessary: Prof. Odej Kao, Prof. Barbara Hammer, Prof. Sven Hartmann, Federico Ponchio, René Weller, but especially Dr. Alexander Hasenfuß and Bassam Mokbel. One person though, Jens Drieseberg, needs his own sentence, in fact he requires his own paragraph:

Jens, I am deeply grateful for all the time we spent, the discussions we had and the ideas we shared. You contributed by far more than you had to and you never let me down. I am lucky, that I had the chance to get to know you and I am very glad that we became close friends.

I would also like to thank all the people and new friends at USI who made moving to Lugano in my final year so much easier. Most notably Dr. Tom Cashman (for the many ideas), Marco (for the coffee), Jochen (for sharing another passion), Parvaz (for the “cube-relationship”), Janine (for being Nina) and Elisa (for the movie nights and the occasional “cheer-up-chats”). Thank you guys for the great time!

Most certainly, I am indebted to my family. Especially to my parents, since they lay the foundation for all I have and also will accomplish in my life. It is very relieving and liberating to know of unquestionable rock-solid backup, if necessary.

Above all, thank you Dana for being part of my life. Your constant support, your exceptional energy and spirit and your motivation encouraged me to successfully pursue this way in the last years. You always ensured that I never lost sight of my goal, especially when it became hazy sometimes.

Furthermore, I would like to thank, Prof. Leif Kobbelt and Prof. Mario Botsch once again, for the valuable ideas and discussions which funneled into the examples in Section 4.3.3.

And finally, I would like to acknowledge that the research presented in Chapter 3 was financially supported by the State of Lower-Saxony and the Volkswagen Foundation, Hannover, Germany.

Contents

Contents **xi**

List of Figures **xiii**

List of Tables **xvii**

1 Introduction **1**
1.1 Contributions 2

2 Background and Related Works **7**
2.1 Surface Reconstruction 9
2.2 Optimization 11
2.3 Interpolation 13
2.4 Compact Representations 17

3 Optimization **21**
3.1 Mesh Massage – A Mesh Optimization Framework 22
3.1.1 Controlling the Triangle Shape 23
3.1.2 Controlling the Distance 24
3.1.3 Combined Optimization 26
3.2 Implementation Details 28
3.2.1 Finding and Maintaining the Correspondence Points 28
3.2.2 How to set up B 30
3.2.3 The Lawson Algorithm 32
3.3 Applications 33
3.3.1 Texture Transfer 33
3.3.2 Remeshing Revised 35
3.3.3 Cross-Parametrization 38
3.3.4 Compatible Remeshing and Connectivity Transfer 39
3.4 Discussion 42

4 Interpolation **47**
4.1 Deformation Gradients and Large Rotations 48
4.1.1 Breadth-First-Search Traversal (BFS) 53
4.1.2 Prioritized Matrix Logarithm Traversal (PML) 57
4.1.3 Prioritized Matrix Logarithm with Normal Difference (PMLN) . . 59
4.1.4 Drawbacks 62
4.2 Multi-Scale Geometry Interpolation 65
4.2.1 Aligning Patches 67
4.2.2 Blending Patches 69
4.2.3 Building the Hierarchy 71
4.2.4 Multiple Input Meshes 73
4.3 Discussion 73
4.3.1 Extrapolation and Constrained Interpolation 77
4.3.2 Timings 78
4.3.3 Robustness 78

5 Compact Representations **83**
5.1 Principal Component Analysis 84
5.1.1 Principal Component Representation 86
5.2 Neural Gas 88
5.2.1 Finding the Interpolation Weights 90
5.2.2 Reconstruction Quality 90
5.2.3 Results 92
5.2.4 Discussion 94
5.3 Exploring Shape Space Alternatives 95
5.3.1 PCA in Different Shape Spaces 96
5.3.2 Discussion 100

6 Conclusion **105**

A Deformation Gradients **109**
A.1 Deformation Gradient Decomposition 112
A.2 Matrix Exponentials 112
A.2.1 Matrix Exponential in 2D 112
A.2.2 Matrix Exponential in 3D 113
A.2.3 Lerp, Nlerp, Slerp? 115
A.3 Interpolation of Matrix Logarithms 116

Bibliography **119**

Figures

1.1 Different triangle meshes, capturing the same geometry 3

2.1 Scanning a polar bear 7
2.2 Structured light scanner 8
2.3 DAVID scanner system 9
2.4 Reconstruction of the bunny model, using Delaunay covering 10
2.5 Poissson surface reconstruction 11
2.6 Mesh optimization: preserving sharp features of the fandisk 12
2.7 The vertex path problem 13
2.8 The vertex path problem for meshes 14
2.9 The problem of large global rotations 15
2.10 Thinning an animation sequence 17

3.1 Applying averaging operations to a mesh 23
3.2 Distances between vertices and their corresponding points 25
3.3 Unconstrained triangle shape optimization 26
3.4 Trade-off parameter between shape and distance optimization 27
3.5 Navigation scheme based on barycentric coordinates 29
3.6 Texture transfer: optimizing triangle distortion for an mesh animation . 34
3.7 Texture transfer: maximum and average distortion 35
3.8 Remeshing: common methods methods 35
3.9 Remeshing: error distribution leads to smaller Hausdorff distance 36
3.10 Remeshing: reduced Hausdorff distance in the presence of PN-triangles 37
3.11 Cross-parameterization: optimizing a horse-to-man sequence 39
3.12 Cross-parameterization: distribution of triangle distortion 40
3.13 Cross-parameterization: maximum and average distortion 40
3.14 Cross-parameterization between the venus head and a skull 40
3.15 Connectivity transfer: create compatible meshes 41
3.16 Optimization: preserving sharp feature lines of the cube 42
3.17 Optimization: preserving sharp feature lines of the fandisk 43
3.18 Different configurations for shape and distance control 44
3.19 Using additional correspondence points 45

4.1 Deformation gradient interpolation: non-intuitive result 48
4.2 Deformation gradient interpolation: difficulties with large rotations . . 49
4.3 Ambiguity of rotation interpolation 50
4.4 Deformation gradient interpolation with breadth-first-traversal 52
4.5 Armadillo without modification 54
4.6 Armadillo modified breadth-first-search traversal 55
4.7 Limitations of a breadth-first-search traversal 55
4.8 False propagation of rotation axes 56
4.9 PML algorithm prevents spreading of wrong rotation axes 58
4.10 Performance of the PML algorithm in the presence of multiple rotations 59
4.11 The PML algorithm for the lion model 60
4.12 Comparison of different traversal methods 61
4.13 PMLN algorithm fails for the helix 62
4.14 Slerp'ed normal path vs. real normal path 64
4.15 Slerp'ed normal and rotation axis vs. real paths for the lion 64
4.16 Slerp'ed normal and rotation axis vs. real paths for the helix 65
4.17 Linear interpolation of a single triangle and a wedge 66
4.18 Interpolation: vertex blending 69
4.19 Interpolation: edge blending 70
4.20 Interpolation: building the patch hierarchy 72
4.21 Interpolation: multiple input meshes 73
4.22 Semantic deformation transfer vs. multi-scale geometry interpolation . . 74
4.23 Interpolation: results for the armadillo, lion and helix 75
4.24 Relative edge and dihedral angle error 76
4.25 Relative volume error 76
4.26 Extrapolation the cylinder 77
4.27 Extrapolating the elephant 77
4.28 Constrained interpolation 78
4.29 Interpolation with additional noise 79
4.30 Interpolating skinny triangles 79
4.31 Hausdorff distance between the remeshed elephant and the original one 80
4.32 Interpolation of highly irregular tessellated meshes. 81

5.1 A compact representation of an animation sequence 83
5.2 Principal component analysis illustrated 84
5.3 Principal components for meshes 86
5.4 Principal Components with average mesh 87
5.5 NG key-frames 89
5.6 Comparing the KG-error 91
5.7 Reconstruction error depending on the number of key-frames 92
5.8 Selected frames from the sequences used for thinning 93

5.9 Reconstruction error for the handstand sequence 97
5.10 Visualization of the root mean square error 98
5.11 PCA in different shape spaces 100
5.12 Comparison between the NG and the PCA approach 101
5.13 Edge space requirements vs. PCA 102
5.14 PCA reconstruction error on a non-optimized sequence vs. optimized sequence 103

A.1 Application of a deformation gradient 109
A.2 Interpolation of deformation gradients 111
A.3 Interpolation of rotation axes 115

Tables

3.1 Remeshing: one-sided Hausdorff distance between the original and the optimized mesh 36
3.2 Remeshing: one-sided Hausdorff distance in case of PN-triangles 37
3.3 Radius ratio and Hausdorff distance 43
3.4 Mesh Massage framework timings 44

4.1 MSGI framework timings 78
4.2 Connectivity transfer: Hausdorff distance between N_t and M_t 79

5.1 Reconstruction error of BNG vs. PCA 91
5.2 Details of the sequences that were used in the section 94
5.3 PSNR error: TMA vs. Key-Probe 94
5.4 Comparing different file formats for storing PCs 99
5.5 Average errors for non-optimized vs. optimized sequences 103

Chapter 1

Introduction

Virtual worlds are safe places. They make our daily life much easier, because they allow us to make mistakes. They let us be explorers in a secure environment. For example, doctors or medical scientists can diagnose a patient in a safe and non-invasive way, aided by digital three-dimensional images of the body.

Art provides another example: consider an ancient vase from an earlier century. It must not be damaged under any circumstances, yet a lot of people are interested in it. We could easily create, in a similar way as for a replica in a museum, a digital copy of this precious piece of artwork and then distribute it safely to interested research groups or simply to connoisseurs around the globe.

Another art form, the one of storytelling, is a little more demanding. The story would be rather boring, if it was all about a single static object like a vase. Instead, whole virtual worlds and universes are invented for the masses. Nowadays, almost every new blockbuster tries to stun the viewer with improved computer generated effects. Games are shelf warmers, if their graphics engine is not capable of new overwhelming and breathtaking 3D effects. But still, the mission of movies and games is to provide safe and entertaining places for the consumer. People can either passively follow the breakneck stunts of a fictional movie character (which no stuntman would survive) or more interactively explore the thrilling artificial environment of a game without risking their own lives to some monster.

Many different research areas that are part of computer graphics provide the fundamental tools for creating these brave new worlds. Of course, the given examples are only the well known eye-catchers, but apart from pure entertainment, countless situations exist where a digital avatar or the simulation of motion is of great use. For example, just think of crash tests or training pilots in a flight simulator.

Over the recent years the results from the different fields of computer graphics added up to form a pipeline for creation and manipulation of digital avatars and worlds. Within the next section we will briefly sketch some of the different stages, that are necessary to create and manipulate digital representations and introduce our

contributions to several important parts of this geometry processing pipeline. Among other things, this pipeline comprises data acquisition, surface reconstruction, optimization, morphing, interpolation and ways of compacting the representation of digital representation of an object.

At the beginning of Chapter 2, we will briefly review some of the most common ways for data acquisition as well as surface reconstruction methods that build upon the acquired data. We proceed with a thoroughly overview of the following stages, integrating our contributions into the context of related works, before we examine the methods in detail in Chapter 3, 4 and 5.

1.1 Contributions

Nowadays, digital copies are mostly acquired by scanning devices, such as 3D laser scanners, structured light scanners, etc. Their common working principle is to sample the object's continuous surface at certain points in space. This creates a set of three dimensional (3D) points, which is often referred to as *point cloud*. Since these points have little explanatory power on their own, they need to be connected or *meshed* in a clever way, such that they represent a closed surface again.

For this purpose the most common method is to take three neighboring sample points and use them as the corner vertices of a triangle. If all sample points are connected this way, they form a so called *triangle mesh* M, which consists of this set of vertices $\boldsymbol{v} = \{v_1, ..., v_n\}$ and a set of faces $\boldsymbol{f} = \{f_1, ..., f_m\}$ where each face stores the vertex indices which indicate a single triangle. The union of all those triangles in M is then a piecewise linear approximation of the real object's continuous surface.

Of course, other forms of polygonal meshes exist, such as quad meshes. But, due to the fact that modern graphics boards (GPUs) are optimized for processing triangles, the majority of surface meshes in computer graphics are triangle meshes.

Yet, the acquired mesh M reflects only *one* of the infinitely many possible representations of the real object's surface, because the sampling of the points from the surface is more or less random.

For example, both meshes shown in Figure 1.1 were acquired from the same bust of Julius Caesar and they represent the underlying geometry equally well. But, the vertices of the right mesh are modified. They were wilfully changed to new 3D positions, such that the triangles became more equilateral. Therefore, the mesh is commonly considered as superior to the one on the left, with respect to its triangle shape.

Mesh Optimization is an essential part of the geometry processing toolbox. As we have seen, infinitely many ways of representing a smooth surface by a triangle mesh exist, as long as this mesh samples the continuous geometry of the object. Hence, we developed a versatile, yet effective framework, which allows to optimize meshes with respect to certain target criteria [WHG08]. At the heart of the method are two control

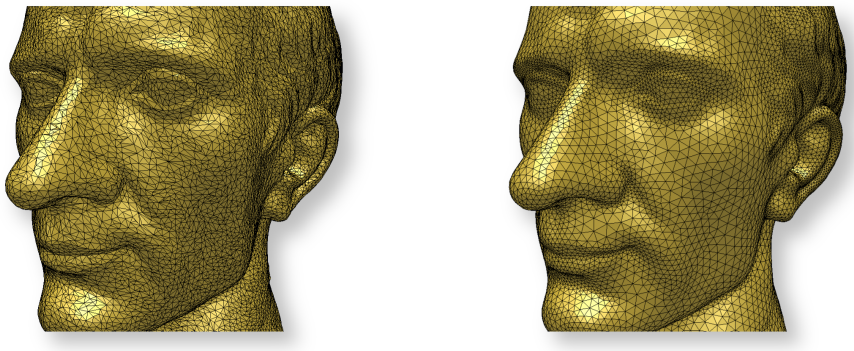

Figure 1.1. Different triangle meshes, capturing the same geometry. Left: original acquired mesh; Right: optimized version of the mesh, with respect to the triangle shape (image courtesy of Aim@Shape Shape Repository).

mechanisms, the first allows to control the triangle shape of the optimized/modified mesh with respect to a reference shape and the second allows to control the distance to some reference geometry. The framework can be applied either as a pre- or post-processing step for a variety of applications and hence improve their results. More precisely, we applied it in the context of texture transfer, cross-parametrization and remeshing methods. Furthermore, it can be utilized to create compatible input meshes for the interpolation between different shapes. The details of the method are described in Chapter 3.

So far, we have only considered surface meshes which are captured from static objects. But what happens if we are also interested in creating and analyzing dynamic scenes or animation sequences of objects? To create the illusion of motion we would need, similar to the pages of a flip-book, a static mesh for every single time step in the animation sequence. The smoother the animation shall be, the more meshes we need.

Unfortunately, this would pose a tremendous and time consuming task to the user, who would have to scan all meshes for an animation sequence and then utilize them in a flip-book-like manner. The problem gets even worse if there is no object to scan, because for the moment, it is only existent as an idea in the user's mind.

This is where specialized 3D modeling software, such as Autodesk 3ds MAX, Blender, Maya and all their clones come to the rescue. If your favorite alien is on vacation or it does not agree with motion capturing for some reason, then this kind of software allows you to create the illusion of motion in a different way: The user designs several meaningful poses of the mesh and the software creates the desired motion by interpolating between these *key-poses*.

Shape Interpolation between triangle meshes is a common building block of many applications. Yet, it is well known that although linear interpolation of the vertices is fast and robust, it will produce rather unexpected results if one of the meshes involved

was subject to large affine transformations, especially rotations. The method presented in this thesis [WDAH10] starts from interpolating the local metric (edge lengths) and mean curvature (dihedral angles) and makes consistent choices of local affine transformations using a global alignment step applied to successively larger parts of the mesh. The local interpolation can be applied to any number of input vertex configurations and due to the hierarchical scheme for generating consolidated vertex positions, the approach is fast and can be applied to very large meshes. It is readily suitable for interpolation between any number of input meshes and therefore permits exploration of the space of shapes that is spanned by certain input poses. We will explain this method in detail in Chapter 4.

So we know how to mimic motion in the digital world. Either by interpolating a set of carefully selected and modeled meshes or we could use techniques like motion capturing, to "copy&paste" the motion of an actor onto the object, assuming that the object is human-like. But such an animation sequence constitutes a rather large amount of data. This raised the question, if it is possible to invert the idea of creating motion through interpolation and extract from a given animation sequence the most prominent meshes, which describe the motion best.

Think of the flip-book analogy again. If we want to create the illusion of a smooth motion we would need many pages with only slightly different drawings. Now assume, we pull out every second page. We could still understand the story, although the animation would be a bit more shaky. A similar idea is utilized in modern 2D video compression methods, several important frames of the movie are selected, which are called *key-frames*. Interpolation between two consecutive key-frames then allows for easy reconstruction of the missing in-betweens and therefore the whole video.

By applying this scheme to a given mesh animation sequence, we could halve the amount of data by deleting every second mesh in that sequence. Later, if we want to reproduce the complete sequence, the missing mesh could be reconstructed by linear interpolation from its former surrounding neighbors. Of course, we can do better, since we are neglecting global information about the whole sequence by simply dropping every second mesh.

Imagine an animation sequence of a person waving good-bye, it is obvious to us that since the person is standing still and only moving the hand from left to right, it would be sufficient to select one key-frame where the hand is on the left side and another one where the hand is on the right side and create all the other frames through interpolation. A more sophisticated thinning scheme which takes the whole sequence into account, could obviously generate better results than just dropping every second mesh.

Compact Representations consider the challenge of reducing the number of meshes that are required to describe the whole animation sequence [WDH$^+$08]. The method is based on a clustering and classification approach that can be used to automatically find the most relevant frames from the sequence. The meshes from the original sequence

can then be expressed as linear combinations of these few key-frames with small approximation error. The key-frames can finally be compressed with any state-of-the-art compression scheme. Overall, this leads to improved compression rates as the number of key-frames is significantly smaller than the number of original frames and the storage overhead for the reconstruction weights is marginal. The detailed description of this method is given in Chapter 5.

Chapter 2

Background and Related Works

Many different ways are practicable to create a digital representation of an object. For example, one could use specialized modeling software as mentioned earlier or one could describe the surface by an implicit formulation and use software like the CGAL 3D Surface Mesh Generator [BO05; RY07]. In these cases the quality of the mesh is often directly guided by the user.

Yet, if a scanning device is used to sample an object's surface, the quality of the generated meshes is closely related to system characteristics of the specific device being used (see Fig. 2.1, 2.2 and 2.3). For example, consider the scanner system depicted on the left side of Figure 2.1, it is basically a coordinate measuring machine (CMM) with a scanning device attached to it and the working principle is briefly as follows:

The scanning device consists of the actual laser and a slightly shifted camera. The projected laser is reflected differently depending on the object's surface and these small deviations are then recorded by the attached camera. Because the arm of the CMM "knows" its position in space it is thus possible to create a point sample of the surface by triangulating the camera position, the laser and the current surface point.

Figure 2.1. Left: polar bear as a real object, recorded by a Kreon 3D scanner system. Right: The digital representation of the polar bear. The three sections of the bear reflect the different stages: 1.) point cloud, 2.) triangle mesh, 3.) shaded surface.

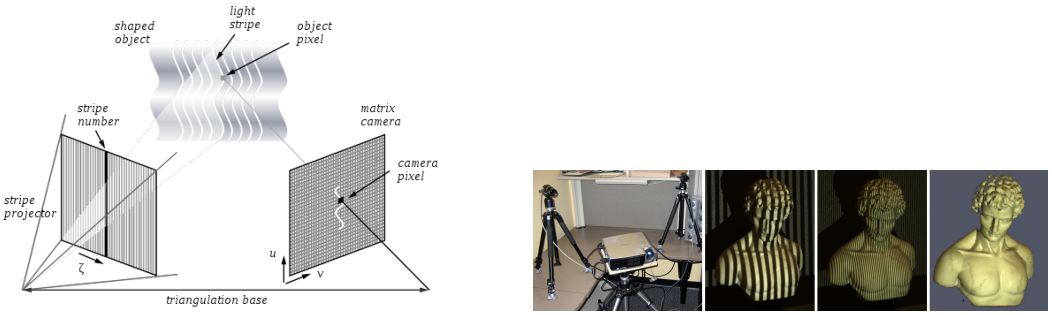

Figure 2.2. Left: stripes are projected onto the surface and their distorted versions are recorded. By analyzing these distortions it is possible to triangulate the points on the object's surface. Right: setup of a structured light scanner, the object and the resulting mesh (picture courtesy of Wikipedia).

For scanning an object, like the polar bear, the user has to manually move the scanner around the object and in this manner create the surface samples. Because the whole system is operated manually, the sampling speed is also directly influenced by the user. Depending on the speed of the scanner moving over the surface, some areas may be sampled less densely than others.

But the main problem arises from the fact that it is not directly possible to scan the object in one pass. On the one hand, if the polar bear is placed on a table, it is easy to scan the upper half, but on the other hand it becomes rather difficult to scan the other side of the bear. Just turning the bear around immediately reveals the next challenge. Although, the belly of the bear is now easy to scan, we create at least one additional point cloud this way.

Therefore, we need to register and align the point clouds against each other, in order to get a single one again which reflects the whole bear. Registration is also necessary if we use a different scanning system as shown in Figure 2.2. Briefly speaking, the structured light scanner takes "pictures" of the object from several different positions in space. Since the pattern of the projected light is known, it is possible to triangulate the surface of the object in a similar way as already described above. But still, we have to face the problem of registration.

Although this problem is solved in theory, in practice one is exposed to several difficulties, such as the overlap in the point clouds, which can approximately double the sampling density or the point clouds cannot be aligned perfectly due to numerical reasons. This would cause some of the points to lie slightly off the real surface and it would be utterly hard to differentiate these "outliers" from actual features.

It does not matter what kind of acquisition method is chosen, raw point clouds have the tendency to be hardly usable, unorganized and outlier ridden. In other words, they are far from being optimal. Thus, surface reconstruction methods, which are part of

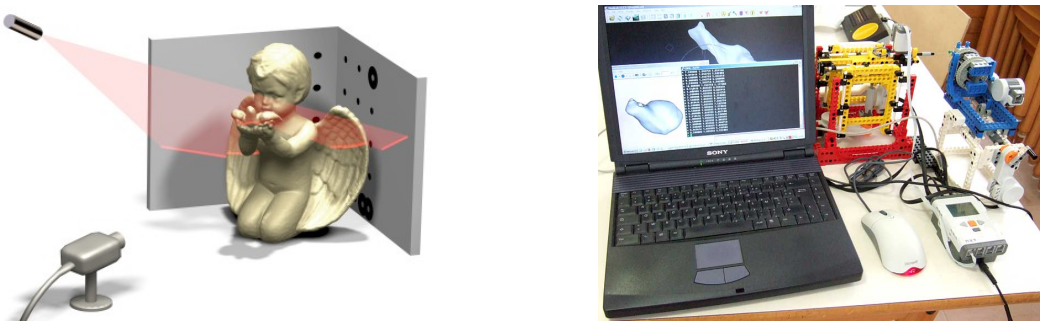

Figure 2.3. Left: The DAVID scanner system which is based on a hand held laser-line scanner and a webcam. Right: a "toy" example of a laser scanner, built from LEGO (picture courtesy of [Hur09; MW09]).

the next stage of the geometry processing pipeline, have to deal with this fact in some way. If they do not, the deficiency of the input point cloud is directly reflected in the reconstructed surface.

2.1 Surface Reconstruction

In the remainder of this section, we will briefly sketch only a few well known approaches, because a full survey on existing methods is well beyond the scope of this section. Instead, we would like to emphasize that the field of surface reconstruction is still an area of active research (e.g. see [MdGD+10]) although many methods have been proposed in the last years. Among the first was the method presented by Hoppe et al. [HDD+92] for converting a point cloud into a triangle mesh.

Other methods use Voronoi diagrams and their duals, since they are well known in 2D as well as in 3D. Although, some of these methods can make certain guarantees about the reconstruction quality or the watertightness of the reconstruction, the main drawback is their sensitivity to noise. Therefore, other approaches were proposed tackling the problem from a different starting point.

In most situations it is easier to deal with the current problem at hand, if it is possible to reduce it from 3D to 2D. Hence, the key idea of Delaunay Covering, proposed by Gopi et al. [GKS00], is to project a 3D point and its k nearest neighbors, into their common least-squares plane, thus creating a local patch and a 2D instance of the problem. In 2D it is now much easier to compute the Delaunay triangulation of the points. The next step is then to use the information of how the points are connected in 2D and simply connect their corresponding points in 3D in the same way. This works surprisingly well, but the resulting meshes have to be optimized and cleaned up. Since it may happen that on the boundary of the patches, the triangulation is

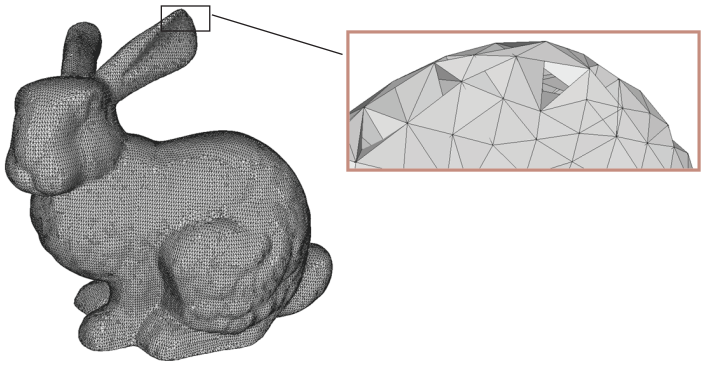

Figure 2.4. Reconstruction of the bunny model, using Delaunay Covering [GKS00]. The resulting meshes need further processing, because the triangulation for each point is only computed locally, which may cause global ambiguities.

different among the contributing patches and an ambiguity in the triangulation, as shown in Figure 2.4 might appear. Amenta et al. [ABK98] were the first to consider the problem of computing the "crust" of a 2D shape by computing the Voronoi diagram and its dual. This approach can also be adapted to 3D as shown by Amenta et al. [Ame99]. Although the cited papers give theoretical guarantees for correctness, the tricky part is –as always– reality. The results of these methods are strongly influenced by noise in the point cloud as well as by the sampling density and therefore require in almost all cases a pre-processing step (de-noising, filtering, smoothing) as well as a post-processing step such as closing holes to make the models watertight. Due to these disadvantages, improved methods were developed, for example Power Crust [ACK01] and Tight Cocone [DG03].

Furthermore, the approach presented by Kazhdan et al. [KBH06] tackles the problem of surface reconstruction from a completely different angle. Here, an indicator function χ is defined, which is 0 outside the object and 1 inside of the object and the surface is reconstructed by extracting an iso-surface. This is rather difficult, because we are lacking the surface to define *inside* and *outside*. But it is not as pointless as it seems. The main observation made by Kazhdan et al. [KBH06] is that a relationship between the set of points and the gradient of the indicator function (see Fig. 2.5) exist. More precisely, they show that the *oriented* points (every point needs an associated normal) can be interpreted as samples of the gradient of the indicator function. In the last stage of the algorithm an adapted variant of the marching cubes algorithm introduced by Lorensen et al. [LC87] is applied to finally extract the surface.

In the remainder of this chapter we proceed with the overview of the geometry processing pipeline that we started in this section with *Data Acquisition* and *Surface Reconstruction*. But we will cover the following stages of *Optimization* (Sec. 2.2), *Inter-*

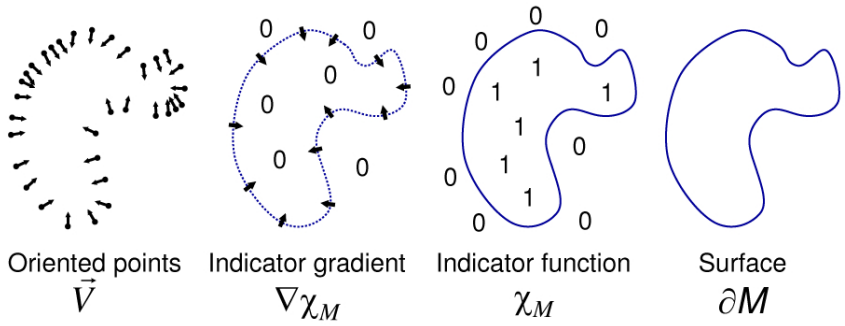

Figure 2.5. From Left to Right: The given input points with normal, the gradient of the indicator function, the approximated indicator function and the reconstructed surface (picture courtesy of [KBH06]).

polation (Sec. 2.3) and *Compact Representations* (Sec. 2.4) more vigilant with respect to the related works clarifying the context of our contributions to those stages.

2.2 Optimization

Over the last decade, triangle meshes have become the most common representation of 3D surfaces in computer graphics and geometric modelling, and there exists an abundance of geometry processing tools for creating, editing, and modifying them. In many applications, however, the result is not optimal and can further be improved by post-processing the mesh. For example, Hoppe et al. [HDD+93] optimize meshes from surface reconstruction, Ohtake and Belyaev [OB01] improve triangulated iso-surfaces, and Surazhsky and Gotsman [SG03] optimize the regularity of a mesh.

The general idea behind mesh optimization is the following: a triangle mesh M is a piecewise linear approximation of some smooth surface S, but there are of course many other meshes that may represent S equally well. Therefore, we can try to modify M such that it is still a good approximation of S, but is better than M in some way (see Fig 2.6). The optimization methods then typically differ in the optimization criterion used and the type of admissible mesh modifications.

For example, the main goal of mesh simplification algorithms (see [Gar99] for a survey) is to reduce the number of triangles while preserving the overall shape. These methods modify the number and positions of the mesh vertices as well as the way in which they are triangulated. Other methods minimize the overall discrete curvature of a mesh [vDA95; DHKL01] and are more restrictive as they change only the connectivity without moving the vertices. In contrast, mesh smoothing algorithms that eliminate noise [Tau95; JDD03; FDCO03] usually optimize only the vertex positions while keeping the triangulation of the mesh fixed.

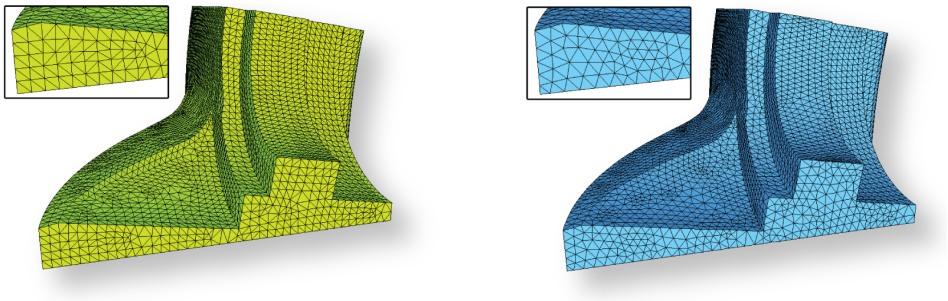

Figure 2.6. We applied the method described in Chapter 3. The initial mesh is shown on the left and the optimized one on the right.

The approach that we present in Chapter 3 falls into the last category in the sense that we also modify only the vertex positions and do not change the connectivity of the mesh. However, we do not have a single optimization criterion, but rather provide a general framework that can be adapted to various settings. The method is somewhat similar to those of Nealen et al. [NISA06] and Liu et al. [LTJW07] as it also uses two competing energies, one to control the shape of the triangles (Section 3.1.1) and another to minimize the distance to some reference geometry (Section 3.1.2). However, our approach can be used for a broader range of possible applications as it differs from [NISA06; LTJW07] in the following way: First, we use mean value coordinates [Flo03] instead of Laplacian coordinates to control the local shape of the mesh triangles, which allows us to also use the triangle shapes from a different reference mesh as target templates (Sections 3.3.1 and 3.3.3). Second, we do not consider distances between the original and modified vertex positions, but more generally minimize the approximate Hausdorff distance between the optimized mesh and some reference geometry. The latter can either be the mesh before optimization or something completely different (Section 3.3.2).

Let us now advance one step further in the geometry processing pipeline and perform the transition from static meshes to dynamic meshes. This transition is somewhat fluent, because we can model a moving mesh as a sequence of many single meshes, which capture a slightly different pose each time. From this point of view we can process an animated mesh or an animation sequence in the same way as a single mesh. We could optimize the whole sequence by optimizing each single mesh (e.g. see Sec. 3.3.1). But what if we do not have a complete set of meshes, that represent the sequence in total, but only a few of them? The next section focuses on the question, how to create motion from just a few static meshes.

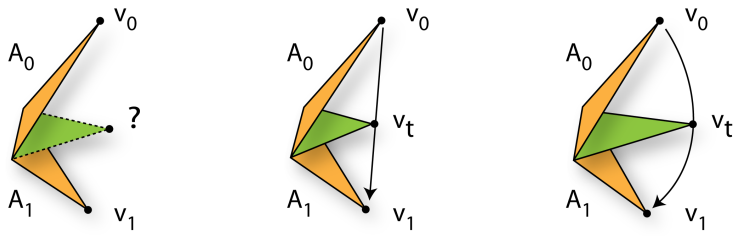

Figure 2.7. Only one vertex is changing its position during interpolation, between the source and the target configuration of the triangle (left). The intermediate triangle as a result of linear vertex interpolation (right).

2.3 Interpolation

Creating vertex positions or trajectories for a mesh from two or more sets of vertex positions is a fundamental building block of many techniques in geometry processing and animation. In the context of morphing [Ale02b] it has been considered a challenge in its own right and is usually referred to as the *vertex path problem*. It is generally accepted that linear interpolation of vertex positions yields undesirable results, because the local shape distorts in the presence of rotations. Consider the following example: Let A_0 and A_1 be two corresponding triangles in a given source and target mesh. They differ only in the position of one vertex, denoted as v_0 and v_1 (see Figure 2.7). Our goal is to compute the position of vertex v_t for every interpolation parameter $t \in [0,1]$, i.e. create the vertex path for v_t. If we simply apply linear interpolation

$$v_t = (1-t)v_0 + tv_1 \tag{2.1}$$

to the vertex positions in this case, we end up with a result similar to the one shown in the middle of Figure 2.7.

In a local context, uncoupled from its neighboring triangles, this yields a plausible result for v_t's path. But, in the same way the vertex path, that is shown on the right side of the picture also represents a plausible path. What is the difference then? Why should we prefer one interpolation method over the other? If we put the triangle back into context, i.e. consider the vertex paths for all vertices of a given mesh, then the difference between both interpolation methods suddenly becomes more pronounced. For the armadillo model shown in Figure 2.8 second from the right we used Eq. (2.1) to compute the vertex paths for every vertex in the mesh and we get an unexpected and even unwanted result due to the shrinking of the triangles, which is caused by linear interpolation.

It is recognizable from the input poses M_0 and M_1, that triangles which belong to the left arm and hand of the armadillo should rotate in a similar manner. Therefore,

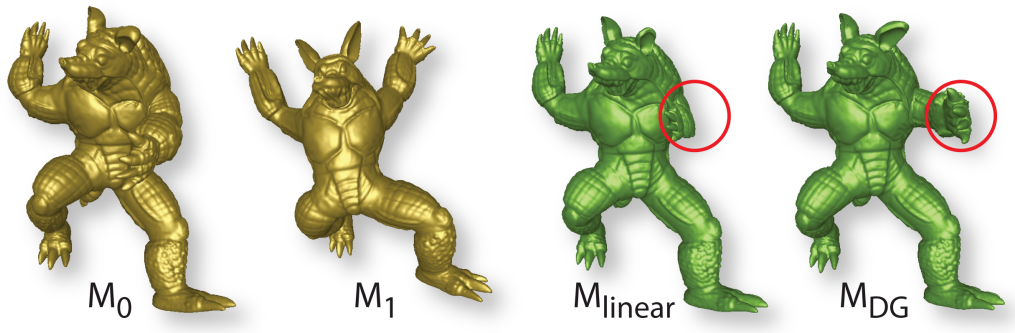

Figure 2.8. Interpolation between M_0 and M_1 using linear vertex interpolation (M_{linear}) and deformation gradients (M_{DG}).

the triangle vertices should change in a common way, i.e. the shape of the triangles is required to stay more or less the same. But, as we have seen, this demand cannot be fulfilled with simple linear interpolation of the vertex positions. Instead the triangles should be interpolated in an as-rigid-as-possible manner [ACOL00], to get a visually more pleasing result as shown on the right side of Figure 2.8.

Better solutions to the vertex path problem, therefore, try to preserve the local shape throughout the interpolation. There are two fundamental approaches that can be traced back to two techniques for morphing planar figures. Sederberg et al. [SGWM93] interpolate intrinsic parameters of a polygon, namely edge lengths and angles. Alexa et al. [ACOL00] identify local affine transformations for each part of the shape and then compute preferred interpolations of these transformations. Both approaches have in common that no global vertex configuration satisfies the local constraints, so that vertex positions are found by an optimization process.

The two approaches have complementary advantages and disadvantages and, as we will see, the disadvantages become more pronounced in the three-dimensional instance of the problem: interpolating edge lengths and angles is robust and fast, but it is difficult to formulate the subsequent optimization problem in such a way that it can be solved both efficiently and uniquely. The fundamental problem is that the orientation of each element of the shape is unknown, meaning that the optimization involves rotations and is non-linear.

On the other hand, prescribing local transformations for each element in the mesh defines this orientation and consolidating the different vertex positions can be done, for example, by linear least squares. But, interpolating the rotational part of each affine transformation is ambiguous. This is mainly due to the non-Euclidean nature of rotations, which makes it impossible to distinguish the *effect* of a rotation by x degrees and another by $x + 360n$ degrees (for any integer n), as they both result in the same *orientation* of the rotated object [Lee08]. Note that picking a preferred rotation (i.e. the one with smallest angle) fails if some parts in the target differ from their corre-

Figure 2.9. Example of a source mesh (left) and a target mesh (right) that illustrates the problem of interpolating large global rotations.

sponding parts in the source by a rotation of more than 180 degrees. In the following we discuss several approaches that apply the idea of factoring and interpolating local affine transformations to geometric modeling approaches in 3D.

Sumner et al. [SP04] were the first to lift the idea of deriving local affine transformations to 3D and then used those *deformation gradients* in [SZGP05] for interpolation among several meshes. For interpolation, they split the transformation matrix into a rotational and a scale/shear part. While the scale/shear part can then be interpolated linearly without any further treatment, the rotational part should be treated in log-space, which requires to compute matrix logarithms and exponentials [Ale02a]. The interested reader is referred to a brief review of deformation gradients in Appendix A.

The main drawback of this approach is that the triangles are interpolated individually, and so a situation as shown in Figure 2.9 cannot be handled correctly. From a global perspective, it is clear that the cylinder is deformed, but for all approaches that are based on deformation gradients [SZGP05; DSP06], it is not possible to distinguish between the top of the cylinder in the source mesh and the rotated top of the bent cylinder in the destination mesh. More precisely, the affine transformation between both parts is a translation. During the interpolation, this part of the mesh will therefore move in a very undesirable way, as it will not rotate at all. The same holds for Poisson shape interpolation [XZWB05], as it was shown by Botsch et al. [BSPG06] that it is equivalent to the deformation gradient approach in this setting.

The fact that deformation gradients rely upon a reference mesh can be exploited to improve the interpolation by using a global alignment in relation to the reference mesh. For example, Baran et al. [BVGP09] achieve this by factoring out the average rotation. But even then it is still not a sufficient representation method for interpolation of large rotations.

What other options do we have then? One idea is to track the rotations during a breadth-first traversal from some seed triangle. In 2D, this actually works pretty well, because all triangles are rotated about the same axis and neighbouring triangles have

similar rotation angles. For example, Alexa has applied this technique to generate the results in [ACOL00]. But in 3D, this approach also requires to propagate the rotation axes, and our experiments show that it is impossible to find a globally consistent distribution of rotation axes unless the object has a very simple shape, and even then the result depends on the traversal order. Moreover, it is often the case that the natural deformation path is not a geodesic in the space of rotations and can therefore not be described correctly by a rotation about a fixed axis [Gra98]. In Section 4.1 we summarize our experiments to compute a consistent traversal order and explain why these approaches are prone to failure.

Another option is to take the connectivity information of the triangle mesh into account, and instead of treating all triangles individually, considering transformations that connect local frames in the mesh. Lipman et al. [LSLCO05] pioneered an approach in this direction. They construct a local coordinate frame for each vertex of the mesh and then consider *connection maps* to encode the transformation between neighbouring frames. A key property of this method is that it represents the local geometry of a mesh in a rotation-invariant way, which appears to overcome the problem that all linear mesh representations (such as deformation gradients and Laplace coordinates) suffer from. Although this may cause counterintuitive results for extreme rotations (by more than 180 degrees) when used as an editing tool, it solves the orientation problem discussed above when used for interpolation. The reason is that the reconstruction process is performed in two steps: the connection maps are used to compute local frames and, based on the local frames, the vertex coordinates are reconstructed.

Kircher and Garland [KG08] improve upon this approach by considering affine connection maps between neighbouring triangles and storing them explicitly. This also results in a two-step linear reconstruction process and appears to handle even large rotations. But the matrices that are involved are rather big (three times the number of triangles) and need to be factorized for each interpolation step, which in turn limits the method to rather small meshes. Baran et al. [BVGP09] reduce the rotation problem by splitting the meshes into patches such that the triangles within a single patch are not rotated by more than 180 degrees relative to the patch frame. This is a significant improvement, however, it is not clear if such a segmentation necessarily exists.

All of the approaches discussed so far try to model the non-linear nature of the problem in a linear way and require only the solution of one or two large but sparse linear systems. Clearly, the problem can also be modelled non-linearly if we accept more involved optimization. Pyramid coordinates [SK04] are a natural non-linear local representation of vertex positions. Kilian et al. [KMP07] define a Riemannian metric that penalizes non-isometric deformations and search for geodesic paths in the resulting shape space. In both cases the reconstruction process is computationally intense. Lastly, another recent method [CL09] with very promising results tries to solve the problem of large rotations by applying a hierarchical version of the mean shift cluster algorithm.

Figure 2.10. A compact representation of an animation sequence can be created by *thinning* the sequence to the most important meshes. Reconstruction of the original sequence is carried out by computing linear combinations of those meshes (sequence courtesy of Baran et al. [BVGP09]).

In contrast, the approach we will present in Chapter 4 follows the idea of Sederberg et al. [SGWM93] and interpolates the local intrinsic properties of the mesh (edges and dihedral angles). In order to derive a globally coherent solution, we utilize a hierarchical shape matching approach. The latter works for much larger meshes than any of the above mentioned techniques, and local intrinsic shape interpolation naturally applies to more than two input configurations. Moreover, our mesh representation actually provides a new kind of shape space and thus has the potential to be used for other applications beyond shape interpolation.

2.4 Compact Representations

For the flip-book example that was used in the introduction (Sec. 1.1, p.2) we had considered the creation of motion by "leafing" through the pages of the flip-book. In other words, from just a few 2D images we could create an animation sequence. Obviously, those few frames are responsible for the final motion that we see in the end. This leads to the immediate question, if it is somehow possible to reverse this process. Given an animation sequence, can we find those frames that reflect the motion inherent in the sequence in the best possible manner?

In general, for storing and transmitting a 3D animation sequence, it should be converted to a compact representation and plenty of sophisticated approaches exist to do so. However, the compression rates can be improved by first *pre-processing* the sequence, reducing it to a smaller number of important *key-frames*. We will show how to find these few key-frames and how to faithfully reconstruct the whole animation via linear interpolation in Chapter 5.

A very simple approach for compressing dynamic meshes is to use *static mesh compression* (see [Ros04], [AG05], and [PKK05] for surveys) on each frame of an animation sequence. This approach is particularly useful in case the connectivity of the

meshes changes from frame to frame. For a compatible sequence, using a static coder on each frame results in poor compression rates since their spatial and temporal coherence is not exploited.

Dynamic mesh compression techniques on the contrary utilize this information. The prediction-based approaches of Ibarria and Rossignac [IR03] and Yang et al. [YKL02] compute the new vertex positions based on the change of the neighbouring vertices in the previous frames of the sequence.

Another option are wavelet-based methods. Guskov and Khodakovsky [GK04], for example, apply wavelets on top of a progressive mesh hierarchy and the resulting wavelet details are encoded in a progressive manner, yielding good results if the input sequence is a rigid-body motion. The goal of the temporal lifting scheme introduced by Payan and Antonini [PA05] is to exploit the coherence in the geometry of successive frames to reduce the information needed to represent the original sequence.

Geometry videos, introduced by Briceño et al. [BSM+03] use a remeshing step to discard the original connectivity information. The main drawback of this approach is its high computational cost.

Another class of compression schemes are clustering approaches. They divide the mesh into subparts and the motion of these subparts is expressed as a set of rigid-body transformations. This idea was first introduced by Lengyel [Len99] and Boulfani et al. [BCAP07; BCA07] later combined clustering with wavelets.

From another point of view, the frames of an animation sequence can be interpreted as the observations of a statistical experiment with the mesh vertices as variables. A well known tool to reduce the information in such a data set is the *principal component analysis* (PCA). The first approach that used PCA for mesh compression was proposed by Alexa and Müller [AM00]. Here the vertex positions in each frame are interpreted as the columns of a matrix. The eigenvectors obtained by singular value decomposition (SVD) of this matrix capture the information inherent to the animation.

The PCA approach is efficient for sequences with small meshes as the matrix size is given by the number of vertices. The main drawbacks of this approach are the computationally challenging SVD and the fact that many animated meshes contain highly non-linear behaviour which is hard to capture globally using this approach. Karni and Gotsman [KG04] extended the PCA approach by coding the eigenvectors with linear prediction. Lee et al. [LKT+07] further showed how to find the optimal number of eigenvectors for a given sequence automatically.

Amjoun and Straßer [AS07] cluster the vertex positions into several local coordinate frames (LCF) and execute a PCA on each LCF with an optimally chosen number of eigenvectors for each LCF. Instead of clustering the vertex positions, Sattler et al. [SSK05] propose to cluster the vertex trajectories in combination with a local PCA. But although this approach scales well with the size of the meshes, it breaks down for extremely long sequences because here the size of the PCA matrix is given by the number of frames.

Although the basic PCA-based approach [AM00] and its improved version [KG04] reconstruct the sequence in the optimal way in terms of the 2-norm, they suffer from the aforementioned drawbacks. Nevertheless, we still advocate that the general idea behind these approaches is promising for a pre-processing step. Consider the flip-book example from the beginning: once we have found the subset of important frames, the key-frames, we can then use linear interpolation again to reconstruct the complete sequence from this subset. And since these key-frames are part of the original sequence, we can also use any of the schemes mentioned above to compress this subset of frames.

A first approach to extract meaningful frames out of captured motion data was presented by Lim and Thalmann [LT01]. It is based on curve simplification and the meaningful frames of the sequence are in fact part of the sequence. This holds as well for the approach by Huang et al. [HCHY05], who propose to solve a constrained matrix factorization problem in a least-squares sense, but since this is an iterative approach it is not very efficient for long sequences. An optimized version was recently presented by Lee et al. [LLWC08]. They first simplify each mesh of the sequence and then use a genetic algorithm (GA) to search for representative key-frames. Although their approach is much faster than [HCHY05], the number of iterations needed by the GA to converge is the bottleneck. Huang et al. [HHS08] discuss how to extract key-frames from 3D video. They reformulate the key-frame selection process as a shortest path problem in a graph that is constructed from a self-similarity map. Since this approach is developed for complete 3D video sequences, the connectivity of the mesh as well as the vertex positions are allowed to change in each time step. This setting, however, is not the focus of our approach that we present in Chapter 5.

Chapter 3

Mesh Optimization

As already pointed out in Chapter 2, the triangulation of a point cloud represents a discrete sampling of the original object's surface. Furthermore, depending on a number of factors, such as the acquisition device, the sampling speed, and the sampling frequency, there are infinitely many possible configurations for such a triangle mesh.

From another point of view, given a triangle mesh, it is possible to modify and optimize the position of the mesh vertices, as long as they remain samples of the object's surface. This teams well with the fact that many geometry processing algorithms can benefit from an optimized input. For instance, the Finite element method (FEM) requires the triangles to be as equilateral as possible to prevent numerical instabilities. For other algorithms the output can further be optimized. This leads to the question, "What is an optimal mesh?" and the straight answer is "It depends..."

In this chapter we present a versatile – yet simple – framework that can be used to either pre-process the input or post-process the results of current geometry processing algorithms and optimize them with respect to various criteria. At the heart of the framework are two mechanisms, one to control the shapes of the mesh triangles by enforcing convex combinations with certain weights, another to constrain the optimized mesh to be geometrically close to a given reference geometry. We can describe the optimization procedure as a variational problem and determine the vertices of the optimized mesh by iteratively solving a sparse linear system. The details of the method are described in Section 3.1 and in Section 3.3 we show the effectiveness of this framework by applying it to several different settings.

Texture Transfer: Suppose we have a set of meshes that constitute an animation sequence and that all meshes have the same connectivity but different geometry. In this setting it is natural to use a single set of texture coordinates for all meshes. If the sequence was generated by hand, e.g. by animating a character, this works out nicely. But if the sequence was extracted from the acquisition data of a real world dynamic scene [AG04], then it may happen that the relative positions of the vertices change from frame to frame, making the texture bounce around. We can use our framework

to drastically reduce this effect by aligning the relative vertex positions in all meshes to those in the first frame. This application is discussed in detail in Section 3.3.1.

Remeshing: Another class of algorithms that take a given mesh M and generate another mesh N that represents the same surface as M, are remeshing algorithms (see [AAGU08] for a survey). Unlike mesh optimization methods, these algorithms do not modify M to get N, but rather compute the new mesh from scratch. Usually, the vertices of the remesh N lie on the triangles of the initial mesh M, thus leading to a rather large Hausdorff distance between both meshes. With our framework, we can roughly halve this distance (see Section 3.3.2).

Compatible Remeshing: represents a specialized variant of remeshing algorithms. Within this setup, the remeshes are desired to have a common connectivity structure, well-shaped polygons, approximate well the input, and respect the correspondence (see [AAGU08]). These properties are often required in the context of morphing between shapes and attributes, multi-model shape blending (see Chapter 4), synchronized model editing, fitting template models to multiple data sets and principal component analysis (see Chapter 5).

Despite the many algorithms, e.g. reviewed by Alexa [Ale02b], that parameterize the meshes over a common base domain, Kraevoy and Scheffer [KS04] as well as Schreiner et al. [SAPH04] described how to construct a mapping from one mesh to another. This *cross-parameterization* can be used, e.g. to transfer attributes and to morph between the two meshes. Both applications rely on the mapping to have as low distortion as possible and we found that our framework can improve the initial parameterizations considerably (see Section 3.3.3).

Connectivity Transfer: is closely related to compatible remeshing. We demonstrate in Section 3.3.4 how to utilize our framework to "copy&paste" the triangle connectivity from a source mesh onto a target mesh. This specific property of our framework is used as pre-processing of non-compatible meshes to convert them into compatible ones. Which can further be used as input for the interpolation method described in Chapter 4.

3.1 Mesh Massage – A Mesh Optimization Framework

In this section we explain the different components of our framework for optimizing a mesh M with vertices $v_1, \ldots, v_m$. The mesh can have arbitrary topology and be with or without boundaries. We first describe how to control the triangle shapes by convex combinations, then discuss how to approximate the Hausdorff distance between M and a reference mesh N, and finally explain how to combine both components.

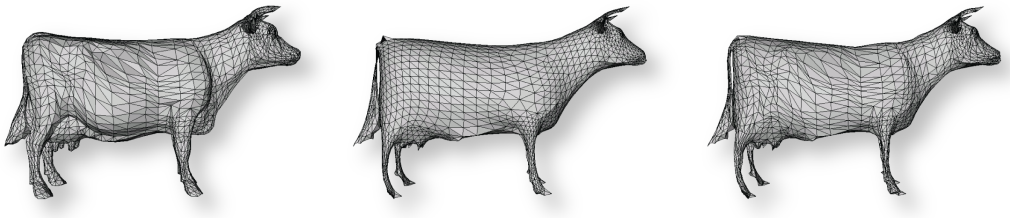

Figure 3.1. Applying averaging operations to a mesh (left) with uniform (centre) and mean value weights (right).

3.1.1 Controlling the Triangle Shape

The basic idea for controlling the shapes of the mesh triangles is to iteratively apply averaging operations to its vertices. Assume that we have for every vertex v_i and each of its neighbours v_j a weight λ_{ij}, then the optimized position $\tilde{v}_i$ of the vertex v_i is given by

$$\tilde{v}_i = \sum_{j \in N_i} \lambda_{ij} v_j, \tag{3.1}$$

where N_i is the index set of all neighbours of v_i. More precisely, we require that all weights λ_{ij} are positive and sum to unity,

$$\sum_{j \in N_i} \lambda_{ij} = 1, \qquad \text{for all } i,$$

so that Eq. (3.1) describes a *convex combination* and hence $\tilde{v}_i$ always lies inside the convex hull of its neighbours v_j, $j \in N_i$.

For example, if all weights λ_{ij}, $j \in N_i$ are identical, then $\tilde{v}_i$ will be located at the barycentre of its neighbours and iteratively applying Eq. (3.1) to all vertices of M will lead to a uniform distribution of points and tends to create equilateral triangles. Instead, if we let λ_{ij} be the mean value coordinates [Flo03] of v_i with respect to its neighbours v_j, then the shapes of the triangles are nicely preserved (see Figure 3.1).

This effect is well known from mesh parametrization, as described in the survey of Floater and Hormann [FH05] and stems from the fact that mean value coordinates are a generalization of barycentric coordinates to arbitrary polygons. A similar effect can be achieved with other barycentric coordinates, but the discrete harmonic coordinates [PP93; EDD$^+$95] may be negative (thus violating the convex hull property) and the Wachspress coordinates [Wac75; MLBD02] are not necessarily well-defined. However, we found in our experiments that the theoretical differences between mean value coordinates and the widely applied discrete harmonic coordinates are more or less negligible in practice (see Figure 3.18, p. 44). Therefore, we focus on mean value coordinates in the remainder of this chapter, keeping in mind that the framework easily allows to switch to other barycentric weighting schemes, if necessary.

Inspired by the deformation transfer method of Sumner and Popović [SP04], we realized that instead of computing the mean value weights from M, we can also compute the weights λ_{ij} from a second mesh N and thus "copy&paste" the shapes of the triangles in N to the triangles in M. Of course, this requires N to have the same connectivity as M, but this setting is less restrictive than it seems and can be found in several applications, two of which we discuss in Sections 3.3.1 and 3.3.3.

Taking averages as described above can actually be seen as repeatedly applying smoothing with either the standard Laplacian (uniform weights) or the discrete Laplace-Beltrami operator (mean value or discrete harmonic weights) [DMSB99]. Unfortunately, this kind of smoothing is well-known to have an undesired shrinking effect on the shape of the mesh (see Figure 3.1), so it cannot be used aggressively unless a complementary mechanism is used to combat the shrinking. We show how to do this in Section 3.1.3.

3.1.2 Controlling the Distance

Besides controlling the triangle shapes, our framework should also allow to control the distance between the optimized mesh M and some reference geometry. In the following we focus on the case that this reference geometry is a second triangle mesh N with vertices $w_1, \ldots, w_n$, but in principle it could be of any kind, e.g. a point cloud, an implicit surface, or a parameteric surface.

Ideally, we would like to measure the Hausdorff distance between the two meshes M and N, but as that is rather costly to compute, we will use the following approximation instead.

Let A and B be two non-empty compact sets. The (minimum) distance from some $a \in A$ to the set B is defined as

$$d(a,B) = \min_{b \in B} \|a - b\|$$

and the one-sided Hausdorff distance $d_{\mathrm{h}}(A,B)$ between A and B is the maximum of all distances,

$$d_{\mathrm{h}}(A,B) = \max_{a \in A} d(a,B). \tag{3.2}$$

The (symmetric) Hausdorff distance $d_{\mathrm{H}}(A,B)$ between A and B is finally defined as

$$d_{\mathrm{H}}(A,B) = \max\{d_{\mathrm{h}}(A,B), d_{\mathrm{h}}(B,A)\}.$$

If A and B are two meshes M and N, then the Hausdorff distance is usually simplified as follows. First, we determine for any point v in M (where v is not necessarily a vertex, but more generally inside one of M's triangles) the corresponding closest point $\hat{v}$ in N (which is again inside one of N's triangles) so that

$$d(v,N) = \|v - \hat{v}\|.$$

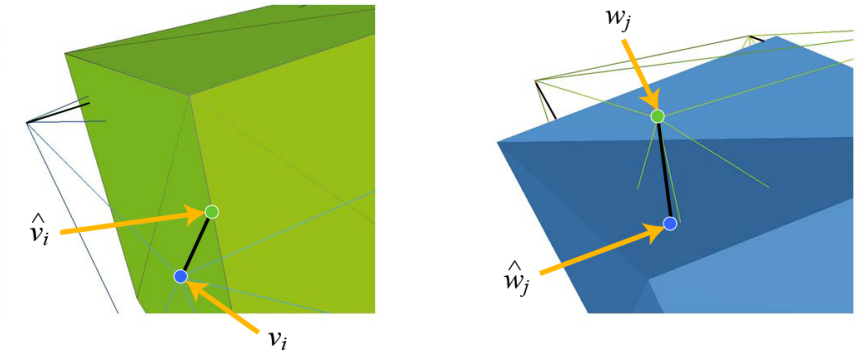

Figure 3.2. Distances between vertices v_i of M (blue) and corresponding points $\hat{v}_i$ in N (green) and vice versa.

In Section 3.2.1 we describe in detail how to find and maintain these corresponding points. Secondly, we do not consider all possible points of M in Eq. (3.2), but only a finite set (usually the vertices v_i of M) and replace the max-norm by the 2-norm,

$$d_{\mathrm{h}}(M,N) \approx \left(\sum_{i=1}^{m} \|v_i - \hat{v}_i\|^2 \right)^{1/2}$$

and similarly for the Hausdorff distance, so that minimizing $d_{\mathrm{H}}(M,N)$ is then simplified to minimizing

$$d_{\mathrm{H}}(M,N)^2 \approx \sum_{i=1}^{m} \|v_i - \hat{v}_i\|^2 + \sum_{j=1}^{n} \|w_j - \hat{w}_j\|^2, \tag{3.3}$$

where $\hat{w}_j$ are the points in M that correspond to the vertices w_j of N (see Figure 3.2).

Although simple to minimize, this approximation of the Hausdorff distance suffers from the fact that the max-norm is replaced by the 2-norm which allows for rather large maximum distances if they are counterbalanced by a lot of small distances. Therefore, we improve the approximation in Eq. (3.3) using a *weighted* 2-norm,

$$\sum_{i=1}^{m} \phi_i \|v_i - \hat{v}_i\|^2 + \sum_{j=1}^{n} \psi_j \|w_j - \hat{w}_j\|^2. \tag{3.4}$$

A theorem by Motzkin and Walsh [MW59] states that there exists a set of weights ϕ_i and ψ_j such that minimizing Eq. (3.4) is equivalent to minimizing the maximum distance

$$\max \left\{ \max_{i=1,\ldots,m} \|v_i - \hat{v}_i\|, \max_{j=1,\ldots,n} \|w_j - \hat{w}_j\| \right\}$$

which in turn is a much better approximation of the Hausdorff distance $d_{\mathrm{H}}(M,N)$ than Eq. (3.3) and can further be improved by considering not only the vertices v_i of M and

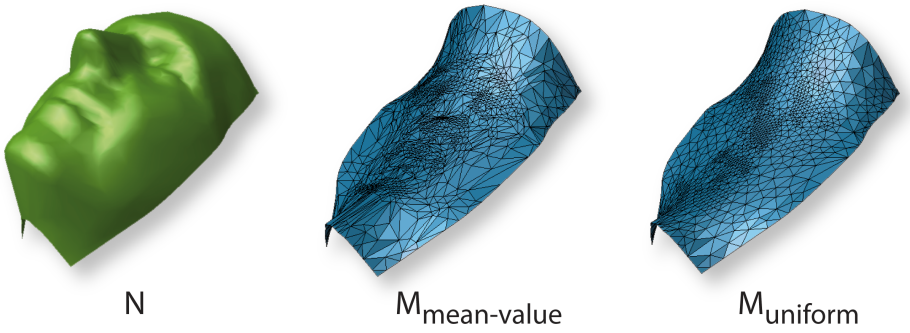

Figure 3.3. Optimizing M solely with Eq. (3.5) using either mean-value weights (middle) or uniform weights (right), results in a minimal surface, spanned between the fixed boundary vertices. Yet, some vertices need to be constrained, otherwise the mesh would collapse onto a single point. Therefore, we counterbalance this shrinking with the approximation of the Hausdorff distance Eq. (3.4).

w_j of N but also additional sample points on the edges and inside the triangles of both meshes. It remains to find the optimal weights ϕ_i and ψ_j, but this can be done iteratively using Lawson's algorithm [Law61] that we describe in detail in Section 3.2.3.

Finally, note that our weighted 2-norm (see Eq. (3.4)), like the approximation in Eq. (3.3), is quadratic in the vertices v_i of M. Thus, minimizing Eq. (3.4) amounts to solving a sparse linear system of normal equations. In Section 3.2.2 we explain in detail how this system is set up. An alternative but less efficient approximation of the Hausdorff distance was presented in [CMP+07].

3.1.3 Combined Optimization

When controlling the triangle shape as described in Section 3.1.1, what we ideally want (instead of iteratively applying the averaging steps) is to place the vertices v_i of M such that the convex combinations Eq. (3.1) are satisfied for all i with $\tilde{v}_i$ replaced by v_i, which is somehow equivalent to doing infinitely many averaging iterations. This amounts to solving the linear system

$$Av = \mathbf{0} \tag{3.5}$$

where $v \in \mathbb{R}^{m\times 3}$ is the matrix storing the x,y,z coordinates for all vertices v_i as columns and A is the sparse Laplacian matrix with elements $A_{ii} = 1$, $A_{ij} = -\lambda_{ij}$ if v_i and v_j are neighbours, and $A_{ij} = 0$ otherwise. The problem is that this system is solved only if all v_i are set to a constant point, which is a rather degenerate solution, but conforms with the remark on the shrinking effect of this kind of smoothing. In order to get a reasonable solution one can constrain the system (3.5) by fixing the positions of a few

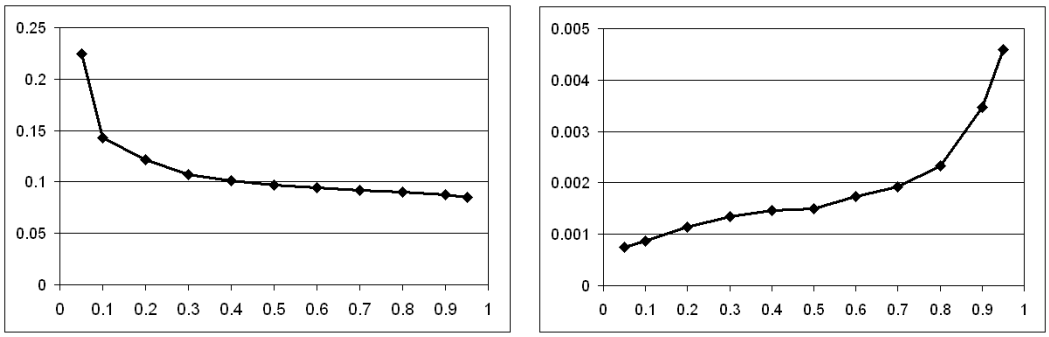

Figure 3.4. Influence of μ on the triangle shape (left) and the Hausdorff distance (right) for the example in Figure 3.14.

vertices (e.g. the vertices at the boundary of a mesh), but this will result in a kind of minimal surface (see Fig. 3.3) spanned between these constrained vertices [PP93]. Again, this is not exactly what we want and we rather choose to constrain it with the Hausdorff distance from Section 3.1.2. Similar to Eq. (3.5), the vertex positions that minimize the approximate Hausdorff distance Eq. (3.4) can also be found by solving the linear system, namely the normal equation

$$B\boldsymbol{v} = \boldsymbol{c}, \tag{3.6}$$

where B is a sparse, symmetric, and positive definite matrix and the right hand side $\boldsymbol{c}$ depends on the fixed vertices $\hat{v}_i$ and w_j. In our framework, we simply take a weighted average of both systems and iteratively solve

$$\big(\mu A + (1-\mu)B\big)\boldsymbol{v} = (1-\mu)\boldsymbol{c} \tag{3.7}$$

with $\mu \in [0,1)$ in order to get the new vertex positions of M. After each iteration the corresponding points and the Lawson weights ϕ_i and ψ_j are updated (see Sec. 3.2.3).

In this approach, μ is a tradeoff parameter between the distance and the shape control, that can be influenced by the user. A small value of μ emphasizes the distance term and leads to very small Hausdorff distances, whereas a larger value of μ gives a better control on the triangle shape at the cost of larger distances (see Figure 3.4). Yet, for all applications shown in Section 3.3 we used the standard setting of $\mu = 0.5$ for an equal weighting between the shape and distance term.

Moreover, we found that performing 10 to 15 iterations of: solving Eq. (3.7), updating the corresponding points and updating the Lawson weights accordingly is usually enough to give nearly optimal results, i.e. neither the distance nor the triangle shapes are further optimized significantly by additional iterations.

Yet, one final remark about Eq. (3.7) has to be made. After sufficiently many iterations the distance between the corresponding points and the mesh vertices (see

Eq. (3.4)) is much less than 1, so all weights ϕ_i and ψ_j in Eq. (3.4) converge to zero and we are losing again the ability to counterbalance the shrinking effect of the shape optimization.

Therefore, we compensate this effect by computing an additional scaling factor σ such that the norms of the shape matrix A and the distance matrix B are equal

$$\sigma\|A\| = (1-\sigma)\|B\| \quad \Leftrightarrow \quad \sigma = \frac{\|B\|}{\|A\| + \|B\|}$$

using the Frobenius matrix norm. We use this norm for the same reason as described by Floater [Flo98], it is relatively fast to compute. Rescaling the matrices this way, causes them to contribute equally to Eq. (3.7) which gave satisfactory results in all our examples. Hence, our method comprises the following steps

- rescale A and B with σ to contribute equally to Eq. (3.7)
- solving Eq. (3.7)
- updating the corresponding vertices $\hat{v}_i$ and $\hat{w}_j$ (Section 3.2.1)
- updating the weights ϕ_i and ψ_j (Section 3.2.3)

Instead of solving Eq. (3.7), we could also minimize the Laplacian system (3.5) and the approximate Hausdorff distance Eq. (3.4) together in a least squares sense, as done in [NISA06; LTJW07], but the resulting normal equation is less sparse than our system (3.7) and takes longer to solve (see Section 3.4).

3.2 Implementation Details

3.2.1 Finding and Maintaining the Correspondence Points

In our framework, the initial corresponding points $\hat{v}_i$ in N for the vertices v_i of M (and similarly the $\hat{w}_j$ for w_j) are set to the closest vertex w_j of N. This requires a global search over all vertices of N, but can be done efficiently by using an appropriate spatial data structure, e.g. a k-d-tree. In this particular case we used ANN [MA97]. Note that this simple strategy requires the mesh M and the reference geometry N to be similar in shape and aligned, which is the case in all the examples that we discuss in Section 3.3.

For a better understanding the following description considers only the corresponding points $\hat{v}_i$, but of course the same applies to all $\hat{w}_j$ as well. Subsequently to the initialization phase, we perform a local search in the triangles around the initial corresponding point $\hat{v}_i$ to find the point closest to v_i among all points in these triangles (see [OWT02] for details) and use it as the correct corresponding point. This closest point usually lies inside one of N's triangles, but can also fall onto a vertex or an edge

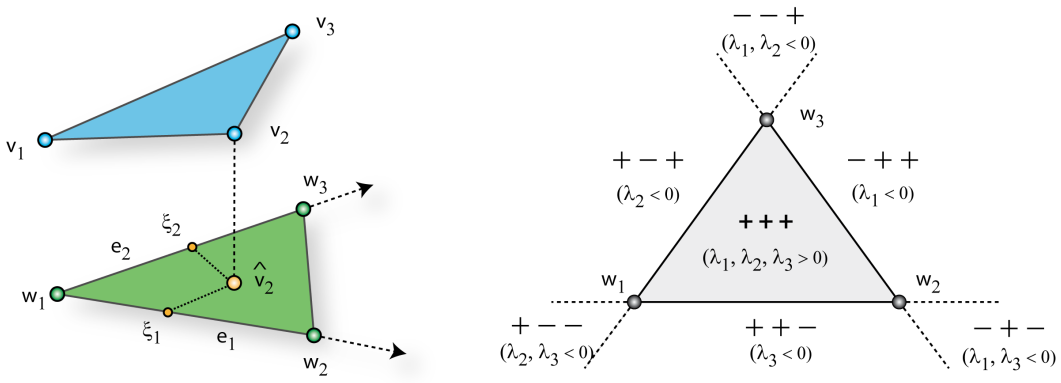

Figure 3.5. The point which corresponds to v_2 is the point $\hat{v}_2$ that has minimal distance to v_2. It can easily be calculated and expressed in local coordinates (ξ_1, ξ_2) w.r.t. the plane spanned by the edge vectors e_1 and e_2 of the green triangle (left). Evaluating the signs of the barycentric coordinates of the corresponding points allows to create a navigation scheme for the update step after each iteration (right).

of N (see Figure 3.5). If we express $\hat{v}_i$ in the plane P that is spanned by the triangle's edge vectors $e_1 = w_2 - w_1$ and $e_2 = w_3 - w_1$

$$P = \{w_1 + \xi_1 e_1 + \xi_2 e_2 | \xi_1, \xi_2 \in \mathbb{R}\}$$

using local coordinates (ξ_1, ξ_2), then finding the corresponding $\hat{v}_i$ simply amounts to minimizing the squared Euclidian distance between v_i and $\hat{v}_i$,

$$D(\xi_1, \xi_2) = \left\| v_i - (w_1 + \xi_1 e_1 + \xi_2 e_2) \right\|^2,$$

with respect to ξ_1 and ξ_2. As usual, this can be done by setting the gradient of D,

$$\frac{\partial D}{\partial \xi_i}(\xi_1, \xi_2) = -2\langle e_i, v - (w_1 + \xi_1 e_1 + \xi_2 e_2)\rangle$$

to zero, for $i = 1, 2$. This can be rewritten as a linear system

$$\begin{pmatrix} \langle e_1, e_1\rangle & \langle e_1, e_2\rangle \\ \langle e_2, e_1\rangle & \langle e_2, e_2\rangle \end{pmatrix} \begin{pmatrix} \xi_1 \\ \xi_2 \end{pmatrix} = \begin{pmatrix} \langle e_1, v\rangle - \langle e_1, w_1\rangle \\ \langle e_2, v\rangle - \langle e_2, w_1\rangle \end{pmatrix},$$

whose solution can be computed directly. Remember, that our initial motivation was to express $\hat{v}_i$ in barycentric coordinates w.r.t N's vertices. In fact, the previous result reveals the desired barycentric weights as $\lambda_1 = 1 - \xi_1 - \xi_2$, $\lambda_2 = \xi_1$ and $\lambda_3 = \xi_2$. The barycentric weights for each $\hat{w}_j$ in M are computed in the same manner.

Local Navigation Now that we are aware of the necessary weights to express each $\hat{v}_i$ as barycentric combination of the triangle's vertices $\hat{v}_i = \sum_{k=1}^{3} \lambda_k w_k$ we can simplify the search for the corresponding vertices in each iteration step significantly. If you think of each barycentric coordinate λ_k as a "force" that pulls $\hat{v}_i$ into the direction of the associated vertex w_k and away from the opposing triangle edge, then it is possible to predict the direction where we find the closest point in the next iteration. More precisely, assume that for some k the weight $\lambda_k < 0$, then we automatically know the corresponding edge and thus the neighboring triangle, where we should look for the next correspondence point. This knowledge allows us to derive the following simple navigation scheme based on the signs of the barycentric coordinates

- if all $\lambda_k > 0$, then we find the closest point inside the triangle
- if one $\lambda_k < 0$, then we identify the corresponding edge and proceed with the neighboring triangle
- if two $\lambda_k < 0$, then we know the closest point has to be among the triangles, which form the "cone" at each vertex (see Fig. 3.5, right). In this case it is very likely, that we have to deal with more than one triangle, therefore we check all triangles from the one-ring of that vertex to find the one that incorporates the corresponding point.

This local search procedure is carried out after each iteration of the algorithm described in Section 3.1.3 in order to always keep the distances between v_i and $\hat{v}_i$ (respectively w_j and $\hat{w}_j$) minimal. Yet, it may happen that for some correspondence points this simple algorithm does not converge. This is caused by corresponding points "flipping" between two adjacent triangles. Usually, such "flipping" between adjacent triangles is prevented by the influence of the shape term of our combined energy (Eq. (3.7)) and we found in all our experiments that the influence of this effect is negligible.

3.2.2 How to set up B

The weighted 2-norm that approximates the Hausdorff distance is quadratic in the vertices v_i of M, because $\hat{v}_i$ and w_j are points in N and thus fixed. Furthermore, any point $\hat{w}_j$ in M can be written as a barycentric combination of the three vertices v_i of M that constitute the triangle which contains $\hat{w}_j$, as previously mentioned in Section 3.2.1. Thus, minimizing

$$d_{\mathrm{H}}(M,N)^2 \approx \underbrace{\sum_{i=1}^{m} \phi_i \|v_i - \hat{v}_i\|^2}_{=:E_1} + \underbrace{\sum_{j=1}^{n} \psi_j \|w_j - \hat{w}_j\|^2}_{=:E_2}. \tag{3.8}$$

amounts to solving a sparse linear system of normal equations. Therefore, we take a quick look how to setup the linear system from Eq. (3.6). The first part E_1 of Eq. (3.8)

measures the distance error from M to N by summing over all weighted distance vectors $\phi_i \|v_i - \hat{v}_i\|^2$. If we define $\boldsymbol{v} = (v_1, ..., v_m)$ and $\hat{\boldsymbol{v}} = (\hat{v}_1, ..., \hat{v}_m)$ we can express this in a much more convenient way as

$$E_1(v) = \sum_{i=1}^{m} \phi_i \|v_i - \hat{v}_i\|^2 = (\boldsymbol{v} - \hat{\boldsymbol{v}})^T \Phi (\boldsymbol{v} - \hat{\boldsymbol{v}})$$

with $\Phi \in \mathbb{R}^{m \times m}$ being the diagonal matrix

$$\Phi = \begin{pmatrix} \phi_1 & & \\ & \ddots & \\ & & \phi_m \end{pmatrix}$$

which contains all weights $\phi_1, ..., \phi_m$, such that we finally have

$$E_1(v) = \boldsymbol{v}^T \Phi \boldsymbol{v} - 2\boldsymbol{v}^T \Phi \hat{\boldsymbol{v}} + \hat{\boldsymbol{v}}^T \Phi \hat{\boldsymbol{v}}.$$

Again, we can find the optimal positions for all v_i by setting the gradient to zero

$$\nabla E_1(v) = 2\Phi \boldsymbol{v} - 2\Phi \hat{\boldsymbol{v}} = 0 \quad \Leftrightarrow \quad \Phi \boldsymbol{v} = \Phi \hat{\boldsymbol{v}} \quad \Leftrightarrow \quad \boldsymbol{v} = \hat{\boldsymbol{v}}. \tag{3.9}$$

In other words the vertices snapped directly onto the positions of their corresponding points, which is no big surprise if we only consider the approximation of the *one-sided* Hausdorff distance E_1. This is similar to the situation described in the context of remeshing in Section 3.3.2, where the vertices of the remesh directly lie on the surface of the triangles of the input mesh.

Yet, this is still not optimal, since the Hausdorff distance is a *two-sided* distance. We also need to take the second part E_2 of Eq. (3.8) into account, which measures the distance from N to M. Similar to the previous case, with $\boldsymbol{w} = (w_1, ..., w_n)$ and $\hat{\boldsymbol{w}} = (\hat{w}_1, ..., \hat{w}_n)$ we can slightly rewrite E_2 as

$$E_2(v) = \sum_{j=1}^{n} \psi_j \|w_j - \hat{w}_j\|^2 = (\boldsymbol{w} - \hat{\boldsymbol{w}})^T \Psi (\boldsymbol{w} - \hat{\boldsymbol{w}}) \tag{3.10}$$

with $\Psi \in \mathbb{R}^{n \times n}$ being the diagonal matrix that contains all weights $\psi_1, ..., \psi_n$. Furthermore, remember that we can express the correspondence points $\hat{w}_j$ through the barycentric combination of the vertices v_i of M, which can be written as

$$\hat{\boldsymbol{w}} = \Lambda \boldsymbol{v} \tag{3.11}$$

with $\Lambda \in \mathbb{R}^{n \times m}$ storing the appropriate barycentric weights for the vertices. We can now replace $\hat{\boldsymbol{w}}$ in Eq. (3.10) with the result from Eq. (3.11) such that

$$\begin{aligned} E_2(v) &= (\boldsymbol{w} - \Lambda \boldsymbol{v})^T \Psi (\boldsymbol{w} - \Lambda \boldsymbol{v}) \\ &= \boldsymbol{w}^T \Psi \boldsymbol{w} - \boldsymbol{w}^T \Psi \Lambda \boldsymbol{v} - \boldsymbol{v}^T \Lambda^T \Psi \boldsymbol{w} + \boldsymbol{v}^T \Lambda^T \Psi \Lambda \boldsymbol{v} \end{aligned}$$

and find the optimum in the same way as described above by

$$\begin{aligned} \nabla E_2(v) = 2\Lambda^T\Psi\Lambda\boldsymbol{v} - 2\Lambda^T\Psi\boldsymbol{w} &= 0 \\ \Lambda^T\Psi\Lambda v &= \Lambda^T\Psi\boldsymbol{w}. \end{aligned} \tag{3.12}$$

Similar to Eq. (3.11), we can also express $\hat{\boldsymbol{v}}$ as barycentric combination of the vertices $\boldsymbol{w}$ of N, which can be written as

$$\hat{\boldsymbol{v}} = \Omega\boldsymbol{w} \qquad \text{with } \Omega \in \mathbb{R}^{m\times n}.$$

This allows us to combine the gradients from Eq. (3.9) and Eq. (3.12)

$$\underbrace{(\Phi + \Lambda^T\Psi\Lambda)}_{=:B}\boldsymbol{v} = \underbrace{(\Phi\Omega + \Lambda^T\Psi)\boldsymbol{w}}_{=:c}$$

and finally reveals the linear system from Eq. (3.6).

3.2.3 The Lawson Algorithm

At the end of Section 3.1.2, on page 24, a theorem by Motzkin and Walsh [MW59] was mentioned which states that, in case of $E_2(v)$, minimizing $\max_{j=1,\dots,n}\|\hat{w}_j - w_j\|$ is equivalent to minimizing $\sum_j \psi_j\|\hat{w}_j - w_j\|_2$ if the proper set of weights $\boldsymbol{\psi} = (\psi_1, \dots, \psi_n)$ is used. The minimum of this quadratic functional can then be found by solving the linear system in Eq. (3.12). The remaining question is, how can we compute these weights?

The idea behind the following approach, that was first utilized by Lawson [Law61], is that vertices with a large distance should contribute more to the weighted 2-norm (see Eq. (3.8)) in the next iteration than those with a small distance.

Lawson's algorithm

1. Set $\psi_1^{(0)} = \dots = \psi_n^{(0)} = 1$ and $r = 0$.
2. Find the optimal $\hat{w}_j^{(r)}$ by solving $\Lambda^T\Psi\Lambda\boldsymbol{v}^{(r)} = \Lambda^T\Psi\boldsymbol{w}$ and setting $(\Lambda v^{(r)})_j = \hat{w}_j^{(r)}$
3. Update the weights,

$$\psi_j^{(r+1)} = \psi_j^{(r)}\|\hat{w}_j^{(r)} - w_j\|, \qquad j = 1, \dots, n,$$

   increment r, and continue with step 2

Although the theoretical convergence of this algorithm to the optimal weights requires infinitely many iterations, we observed that in practice a few iterations (5 to 10) usually suffice to get very close to the optimum. Please note that although, not immediately visible, the same holds for the gradient of $E_1(v)$ in Eq. (3.9), but here the matrix

Λ is simply the identity. This is due to the fact, that we assume the correspondences from M to N to always start at a vertex v_i of M. In other words, expressed in terms of barycentric combinations of the triangle vertices, one of the weights is always 1, but as soon as we drop this restriction and allow the origin of the correspondence to lie inside a triangle of M, we have again the same form of the linear system as in Eq. (3.12). Therefore, the iterative Lawson algorithm teams up very nicely with the character of the framework, since we need to iteratively update the corresponding points anyway (Sec. 3.2.1).

3.3 Applications

This section examines several applications where we successfully applied our framework either as pre- or post-processing method. Beginning in Section 3.3.1, we optimize a mesh animation sequence and underline the framework's flexibility in the context of remeshing in Section 3.3.2. The benefit of post-processing cross-parameterizations is demonstrated in Section 3.3.3 and finally we show how to create compatible input meshes for the interpolation method that shall be described in Chapter 4.

3.3.1 Texture Transfer

Suppose we have a mesh M_1 that approximates some surface S. If we now move the vertices of M_1 slightly in the local tangent planes, then we get a mesh M_2 that also approximates S and looks almost identical to M_1 if rendered with flat or smooth shading. However, if we texture both meshes and use the same texture coordinates for corresponding vertices, then we will clearly see the difference as the shapes of the triangles in M_2 are no longer similar to those in M_1, thus leading to texture distortion.

These kinds of artifacts occur, e.g. when a compatibly meshed sequence of meshes is extracted from some unstructured acquisition data of a dynamic scene. Such an approach has been described by Annuar and Guskov [AG04]. They start with an initial template mesh that is then propagated through the frames of the animation, based on an adaptation of the Bayesian multi-scale differential optical flow algorithm. Since the flow is invariant to motion in the tangent plane, the artifacts described above tend to occur and as a result the texture seems to wobble over the animated mesh during the sequence. The top row in Figure 3.6 shows the texture distortion for several meshes M_i from the dancing man sequence in [AG04] with blue signifying low and red indicating high distortion. The distortion is measured with respect to the first mesh M_1 of the sequence and using the symmetric maximum shear,

$$\max\left\{\sigma_1 + \frac{1}{\sigma_1}, \sigma_2 + \frac{1}{\sigma_2}\right\} - 2, \tag{3.13}$$

as a distortion measure. More precisely we consider for each triangle in M_1 the linear map to the corresponding triangle in M_i and compute the singular values σ_1 and σ_2 of

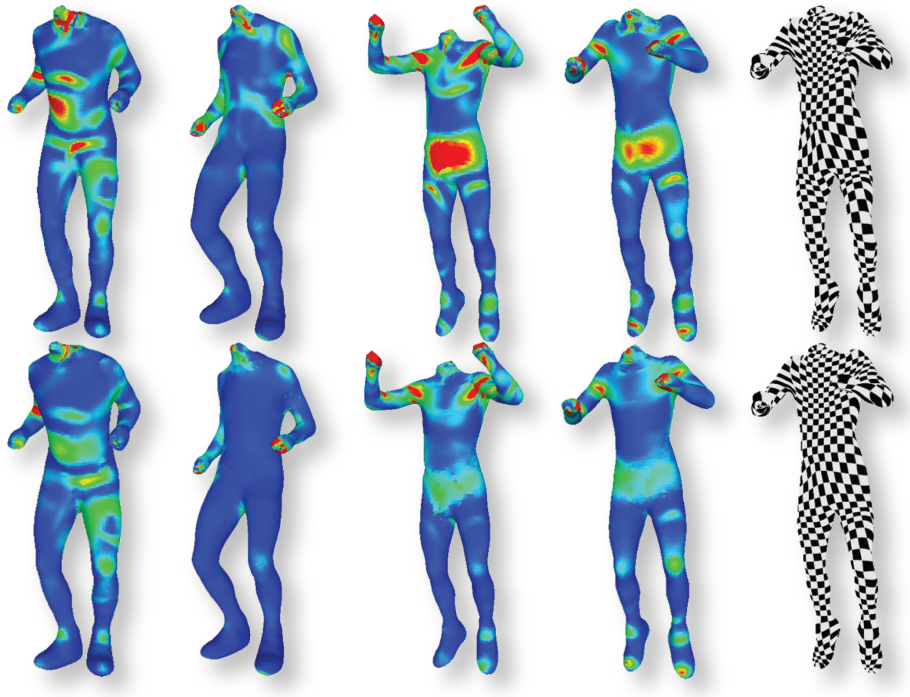

Figure 3.6. Texture distortion for several meshes selected from the animation sequence of a dancing man and textured version of the last mesh. Top: original meshes, bottom: optimized meshes.

this mapping, which capture the distortion of this triangle in the principal directions. In order to penalize over- and undersampling in the same way, we symmetrize these values by adding their inverse, so that shrinking by a factor of $1/2$ is considered equally bad as expanding by a factor of 2, and finally take the worse of both values.

We can apply our framework to optimize the meshes M_i in the sequence by taking M_i itself as the reference geometry and using M_1 as the template for the triangle shapes, i.e. we use the mean value weights λ_{ij} that were computed from M_1 to optimize M_i. As a result, the vertex positions in the flat surface regions are corrected and the texture is fixed throughout the sequence. This results in improved visual quality and is shown in the bottom row of Figure 3.6. The rightmost column shows the last mesh again, but this time textured to emphasize the influence of the distortion to the texture.

Figure 3.7 further shows the maximum and the average distortion for all meshes in the sequence and confirms that our method is able to reduce both for all meshes. Interestingly, the peaks in the average distortion plots correspond to the jumps of the

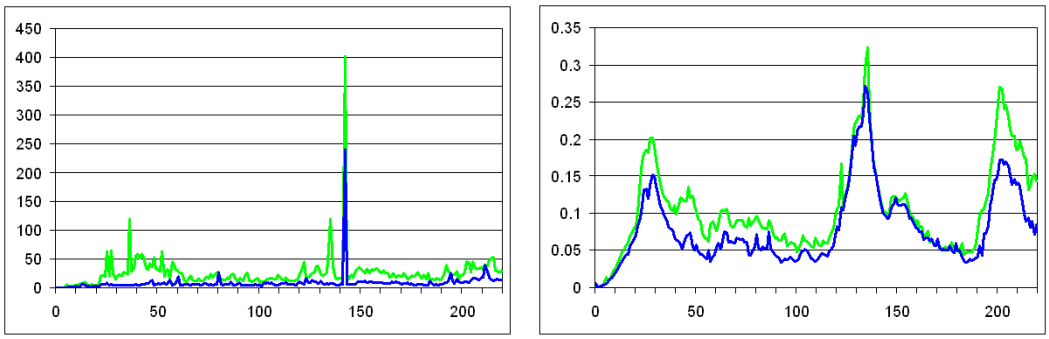

Figure 3.7. Maximum (left) and average (right) distortion for all meshes in the original (green) and the optimized (blue) dancing man sequence.

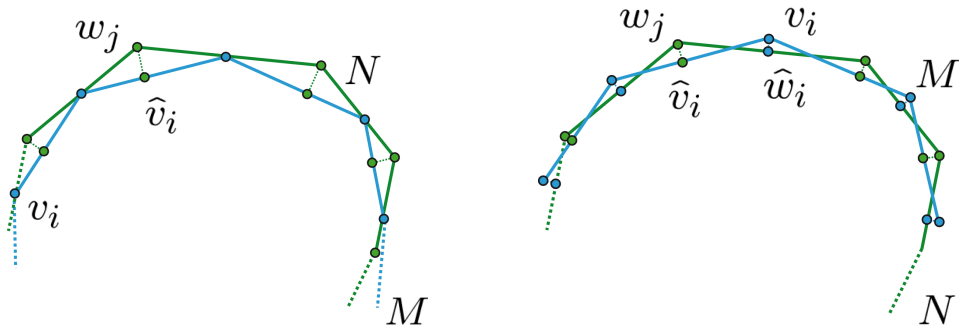

Figure 3.8. Left: Common remeshing algorithms cause the vertices of the newly created mesh M (blue) to lie on the surface of the input mesh N (green), which leads to a rather large Hausdorff distance. Right: Applying our framework in this setting leads to a much better error distribution, thus approximately halving the distance.

dancing man, where the triangle distortion tends to increase due to the extreme global shape deformations in the mesh.

3.3.2 Remeshing Revised

Remeshing is an essential part of the geometry processing toolbox. Although the output generated by modern acquisition tools like 3D scanners captures 3D objects well in terms of geometry, the quality of the generated mesh is often bad with respect to the triangle shapes or the connectivity. To compensate these disadvantages, many algorithms have been proposed to generate for a given source mesh M_1 a remesh M_2 that has the same shape but is superior, e.g. by having a more uniform or curvature-adaptive sampling density, better triangle shapes, or more regular vertex connectivity.

Figure 3.9. The top row reflects the situation sketched on the left of Figure 3.8. Some vertices of the remesh M_2 (top right) live on the triangles of M_1 (top left), thus causing almost no error from M_2 to M_1. But in the other direction, the error from M_1 to M_2 is rather large. Our framework is able to reduce the overall error in the two sided Hausdorff distance significantly in this setting, as the bottom row as well as Table 3.1 confirms. The error between M_1 and M_2 is much more evenly distributed.

	max	mean	RMS
$M_1 \rightarrow M_2$			
original	0.605037	0.010509	0.017324
optimized	0.258184	0.010494	0.014429
$M_2 \rightarrow M_1$			
original	0.256721	0.009208	0.014566
optimized	0.206491	0.010114	0.014120

Table 3.1. One-sided distances between the source mesh M_1 and the original and the optimized remesh M_2 for the hand model in Figure 3.9.

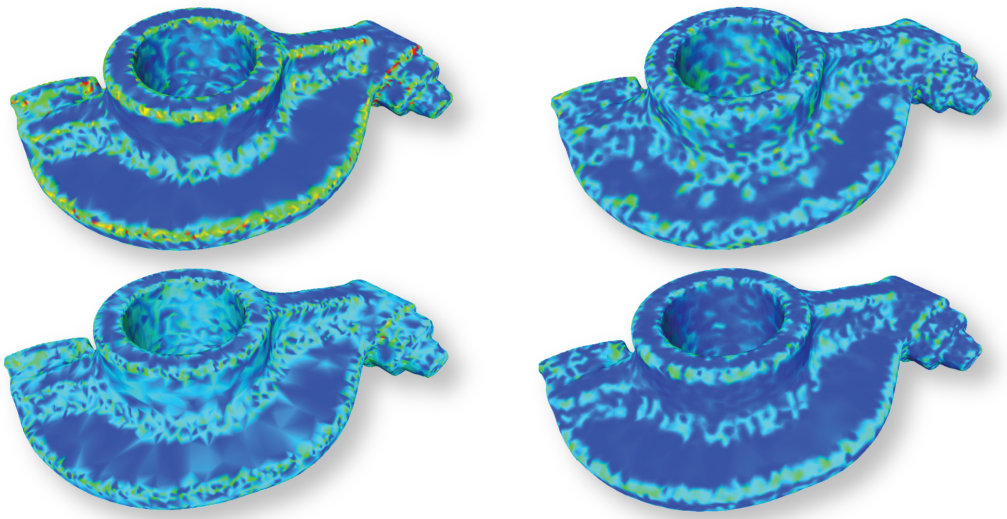

Figure 3.10. Optimization of the rockerarm: one-sided distances between the source mesh M_1 (left) and the remesh M_2 (right) for the original (top) and the optimized (bottom) remesh. Here, the vertices of the remesh lie on the PN-triangles over the source mesh.

	max	mean	RMS
$M_1 \to M_2$			
original	0.046016	0.001729	0.002432
optimized	0.022029	0.001508	0.001970
$M_2 \to M_1$			
original	0.020577	0.001572	0.002185
optimized	0.024118	0.001411	0.001882

Table 3.2. One-sided distance between the source mesh M_1 and the original and the optimized remesh M_2 for the rockerarm model in Figure 3.10.

In almost all remeshing methods, the vertices of the remesh M_2 lie on the triangles of the source mesh M_1, which is far from optimal regarding the Hausdorff distance between both meshes. An exception is the approach of Surazhsky et al. [SAG03] that places the vertices of M_2 on interpolating PN-triangles over M_1.

However, in both cases we found that we can use our framework to reduce the Hausdorff distance by 50% and more. Note that this setting does not require M_1 and M_2 to have compatible connectivity because we do not want to “copy&paste” the triangle shapes from the source mesh to the remesh. Instead, M_1 serves only as the refer-

ence geometry and M_2 itself as its triangle shape reference, i.e. we compute the mean value weights λ_{ij} from the same mesh that we also optimize. Figures 3.9 and 3.10 show the results of the optimization process by colour-coding the one-sided distances between the source mesh and the original as well as the optimized remesh and Tables 3.1 and 3.2 list the numerical values. The vertices of the remeshed hand model were sampled from the triangles of the source mesh and we can see that this gives a relatively small distance from the remesh M_2 to M_1 but a relatively large distance in the other direction, and thus a large Hausdorff distance. By optimizing the mesh we distribute the distance more evenly and thus reduce the Hausdorff distance by about 60%. This is reduced to 50% in the example of the rockerarm, where the vertices of the remesh lie on the PN-triangles over the source mesh. All distances were measured with the Metro tool [CRS98].

3.3.3 Cross-Parametrization

We can also apply our framework to reduce the distortion of cross-parametrizations, i.e. mappings between two meshes with different shapes but compatible connectivity. In particular we used our method to optimize the horse-to-man morphing sequence and the cross-parameterization between the venus head and the skull model, where both examples were generated by the algorithm of Kraevoy and Sheffer [KS04].

The setup for our framework is similar to the one used in the previous Section, i.e. the horse and the venus head are used as reference meshes for the triangle shapes, while the reference geometry is provided by the same mesh that is also optimized (the mesh of the man and the skull model, respectively).

Figure 3.11 visualizes the distortion of the cross-parameterization between the first and the last mesh of the horse-to-man morphing sequence for the original (top) and the optimized mesh (bottom) in two ways: by colour-coding the distortion Eq. (3.13) per triangle and by showing how the parameterization maps a regular texture from the first to the last mesh. Both plots show that our framework successfully reduces the distortion, which is also confirmed by the clear shift towards small values in the histograms of the distortion distribution shown in Figure 3.12.

Figure 3.13 further shows how the maximum distortion is reduced during the iterations of our optimization process. Note that this comes at the price of a slight increase in average distortion. Figure 3.14 finally shows the distortion of the parameterization between the venus head and the skull model before and after our optimization.

Figure 3.11. Left: shaded and textured version of the first frame from the horse-to-man sequence that serves as a reference frame for the triangle shape. Right: distortion of the cross-parameterization between the first and the last frame of the sequence before (top) and after (bottom) optimization.

3.3.4 Compatible Remeshing and Connectivity Transfer

This setting is closely related to the previous section, but instead of computing a common base mesh or a cross-parameterization, we strive to find an easy way to "copy&paste" the connectivity from one mesh to another, thus making them compatible. This specific application of our framework is more or less a proof of concept, we utilized it, as described in the following, to test the impact of a drastically changed connectivity on the interpolation method introduced in Chapter 4. Suppose we have two reference meshes N_1 and N_2 with vertices $\boldsymbol{w}^{(1)}$ and $\boldsymbol{w}^{(2)}$ which share the same connectivity, as shown in the upper row of Figure 3.15. We would like to generate

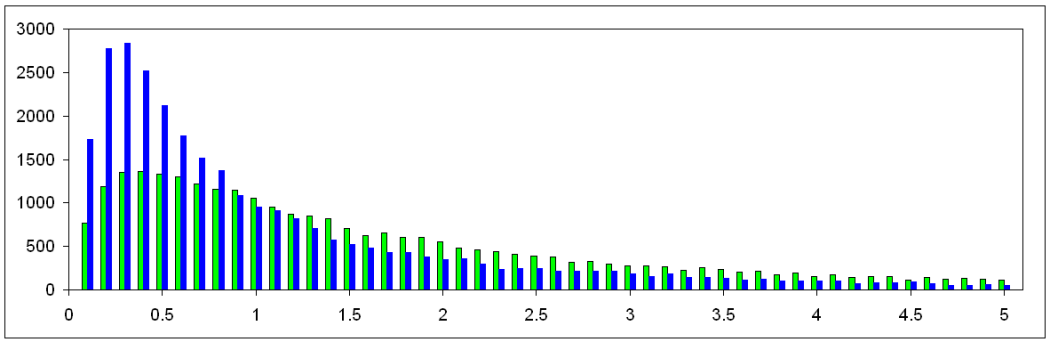

Figure 3.12. Distribution of the distortion per triangle in the horse-to-man sequence before (green) and after (blue) the optimization.

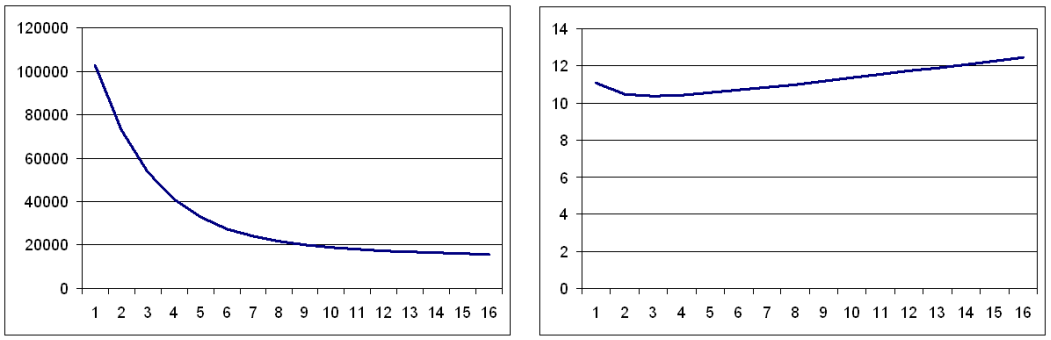

Figure 3.13. Maximum (left) and average (right) distortion over 16 iterations of the optimization process for the horse-to-man sequence.

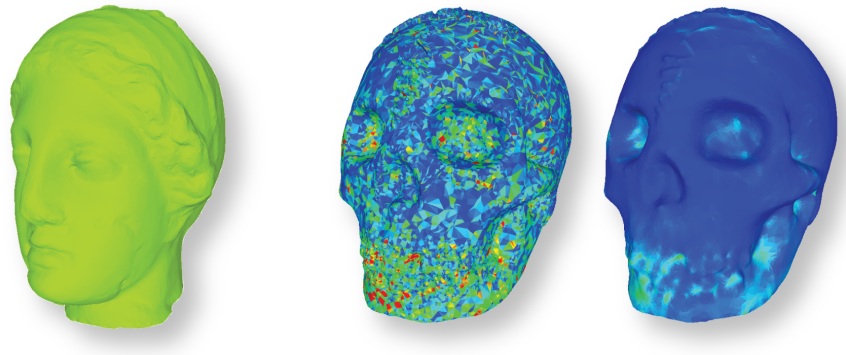

Figure 3.14. Cross-parameterization between the venus head and a skull. The venus model (left) provides the shape reference. The target model without modification is shown in the middle. With optimization (right) the distortion in the skull is reduced significantly.

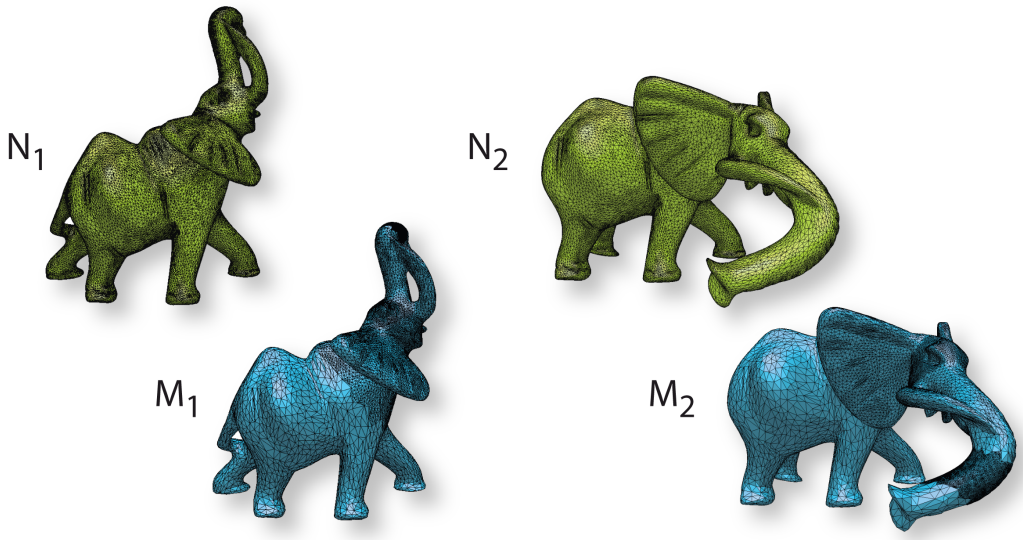

Figure 3.15. The reference meshes N_1 and N_2 shown in green in the top row share the same connectivity. M_1 approximates the same geometry as N_1, but we changed its connectivity significantly by applying several Catmull-Clark subdivision steps, as well as some quadric edge collapses to certain areas of the mesh. This new connectivity is then transferred to M_2, thus creating a new set of meshes that capture the same poses as N_1 and N_2, but with a significantly different triangle connectivity.

a second set now, including mesh M_1 with vertices $\boldsymbol{v}^{(1)}$ and mesh M_2 with vertices $\boldsymbol{v}^{(2)}$. For this new set we have the following requirements: M_1 and M_2 must be compatible, they must capture the same geometry as N_1 and N_2, but they should have a completely *different connectivity*. For example, M_1 could be generated as the output of some remeshing algorithm or as in this particular case, it is the very same mesh as N_1, but with a significantly scrambled triangle connectivity.

More precisely, the elephant shown in Figure 3.15 was created by applying several subdivision steps to certain parts of the trunk and the area around the head. In other regions of the mesh we reduced the triangle density by applying several quadric edge collapse steps. But how do we create a compatible M_2?

Given N_1, N_2 and M_1 this can easily be achieved with our framework. In the first step we use N_1 as the reference geometry and compute the corresponding points between N_1 and M_1. In other words, for any vertex $\boldsymbol{v}^{(1)}$ in M_1 we know the corresponding closest point $\hat{v}^{(1)}$ in N_1, which can be written as a barycentric combination of the vertices $\boldsymbol{w}^{(1)}$ of N_1 in the following way,

$$\hat{v}^{(1)} = \sum_i \lambda_i w_i^{(1)}.$$

Thus we have, briefly speaking, projected the connectivity of M_1 onto the surface of

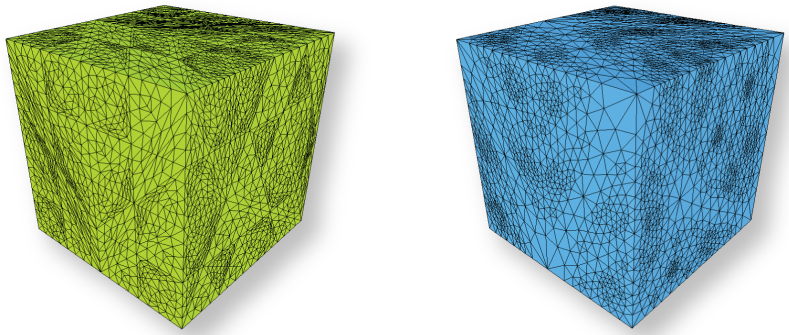

Figure 3.16. This example emphasizes how well sharp feature lines of the initial mesh (left) are preserved in the optimized mesh (right). The approximate Hausdorff distance always pulls the vertices close to the feature lines. In this setting we used uniform weights for triangle shape optimization.

N_1. In the next step, we simply substitute the triangle vertices of N_1 with the vertices of N_2, such that

$$\hat{v}^{(2)} = \sum_i \lambda_i w_i^{(2)}$$

which yields the projection of the connectivity of our –yet to be constructed– M_2 on the surface of N_2. Finally, we create M_2 by exploiting the connectivity information from M_1 and using the $\hat{v}^{(2)}$ as initial positions for the vertices $v^{(2)}$. We can further optimize the result and take N_2 as the reference geometry and optimize M_2 w.r.t. the shapes of M_1's triangles, as shown in Sec. 3.3.2.

3.4 Discussion

We showed that our framework is simple and flexible enough to be applied to a wide range of applications. For example, the methods of Nealen et al. [NISA06] and Liu et al. [LTJW07] cannot be used for optimizing remeshes (see Section 3.3.2) as they can handle only meshes with compatible connectivity. The advantage of [NISA06; LTJW07] is that they need to solve only one $n \times n$ system, whereas we have to do this iteratively (10 to 15 times).

However, our system can be solved more efficiently because it is sparser and if additional constraints (e.g. feature preservation) are required, then [NISA06; LTJW07] need to solve a $3n \times 3n$ system, which is as expensive as our iterative approach. Still our method is able to preserve features well, as shown in Figures 3.16 and 3.17. In these examples, we optimized the given mesh (left) using uniform weights λ_{ij} and the

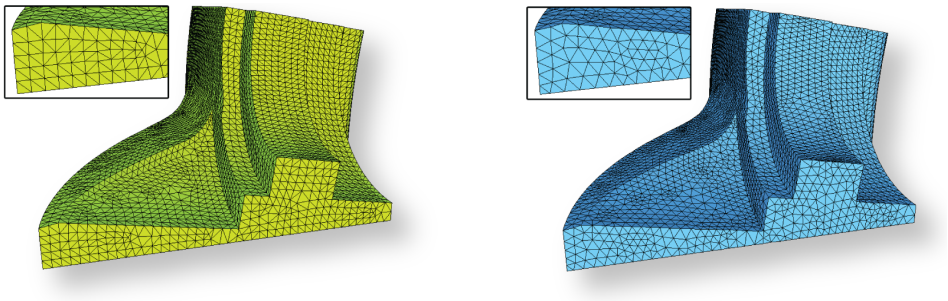

Figure 3.17. Initial mesh (left) and optimized mesh (right).

	min	mean	distance
original	0.3235	0.8505	
lin. + tplane [NISA06]	0.2770	0.9089	$1.87 \cdot 10^{-3}$
lin. + red. Laplacian [NISA06]	0.1525	0.9095	$1.98 \cdot 10^{-4}$
Mesh Massage	0.3700	0.9434	$1.06 \cdot 10^{-3}$

Table 3.3. Radius ratio and Hausdorff distance for the fandisk model in Figure 3.17.

given mesh itself as the reference geometry. During the optimization, the features of the mesh are nicely preserved, because the approximated Hausdorff distance Eq. (3.4) always pulls the vertices close to the feature lines. But apart from this constraint, all vertices are free to move and yield very uniform triangle shapes (right).

Table 3.3 compares our results to those of Nealen et al. [NISA06]. Although their linear reduced Laplacian approach gives slightly smaller distances, the triangle shapes (measured by the radius ratio, i.e. twice the ratio of the circumradius to the inradius of the triangle) are worse because they need to fix the feature vertices in order to preserve the mesh features. If we use our framework for this kind of shape optimization on the armadillo dataset, we achieve better results than [NISA06] both in terms of distance ($2.45 \cdot 10^{-3}$ vs. $2.63 \cdot 10^{-3}$) and triangle shapes (min: 0.112 vs. 0.091, mean: 0.892 vs. 0.868). The most time consuming parts of our framework are the factorization of the system (3.7) using TAUCS [Tol03] and updating the corresponding points. The timings in Table 3.4 were measured on an Intel Core2 Duo E6400 with 2GB of RAM.

One potential drawback of our framework is the calculation of the corresponding points. If the distance between the meshes is too large, the corresponding points cannot be determined correctly anymore.

As stated at the beginning of this chapter, the framework allows to utilize different barycentric weighting schemes for the triangle shape control. The graph on the right side of Figure 3.18 compares the widely used discrete harmonic (or cotangent) weights

	vertices	factorization	update
cube	3860	0.07 s	0.21 s
fandisk	6475	0.18 s	0.40 s
rockerarm	10044	0.25 s	0.48 s
dancing man	15830	0.46 s	0.93 s
armadillo 17k	17297	0.44 s	0.98 s
horse-to-man	17489	0.48 s	1.00 s
venus-to-skull	23908	0.71 s	1.10 s
hand	147634	14.70 s	7.69 s
armadillo	172974	16.14 s	10.36 s

Table 3.4. Run times for one iteration.

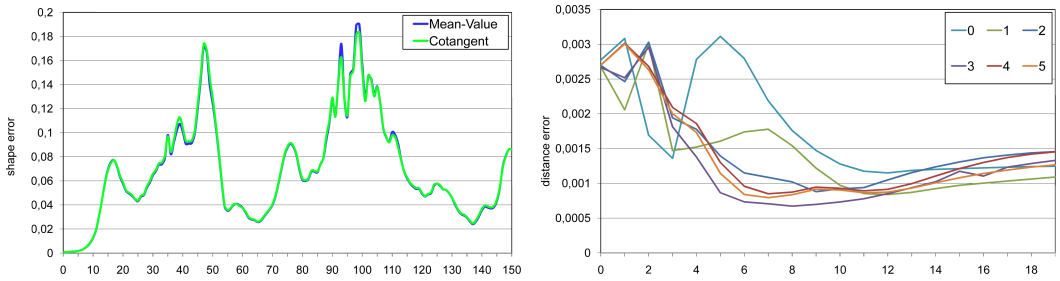

Figure 3.18. Left: The influence of using cotangent weights instead of mean-value weights for optimization (see Sec. 3.1.1). In this case we exemplarily show the results for the sequence in Figure. 5.1 on page 83, since the behavior for the other sequences is comparable. Right: Using additional correspondence points for the example shown in Figure 3.19 and their impact on the approximated Hausdorff distance. Each curve reflects a midpoint subdivision step, executed per mesh triangle.

with our choice of the mean value weights. Although the cotangent weights might become negative, all our experiments showed that in our setting they perform more or less equivalent to the mean value weights as the right side of Figure 3.18 emphasizes.

Since our framework is able to handle additional correspondence points on edges or within triangles besides the mesh vertices, the left graph in Figure 3.18 demonstrates this possibility and the influence on the approximated Hausdorff distance. Each curve is related to a midpoint subdivision step. In this setting we optimized the cube (18k faces) shown in Figure 3.19 while the reference geometry was provided by a sphere (5k faces). Using the standard setting of our framework and taking only the triangle vertices (respectively 0 subdivision steps) and their corresponding points into account, the mesh is able to “float” more freely over the reference geometry as emphasized by the increased error around the 5th and 6th iteration of the algorithm. On the contrary,

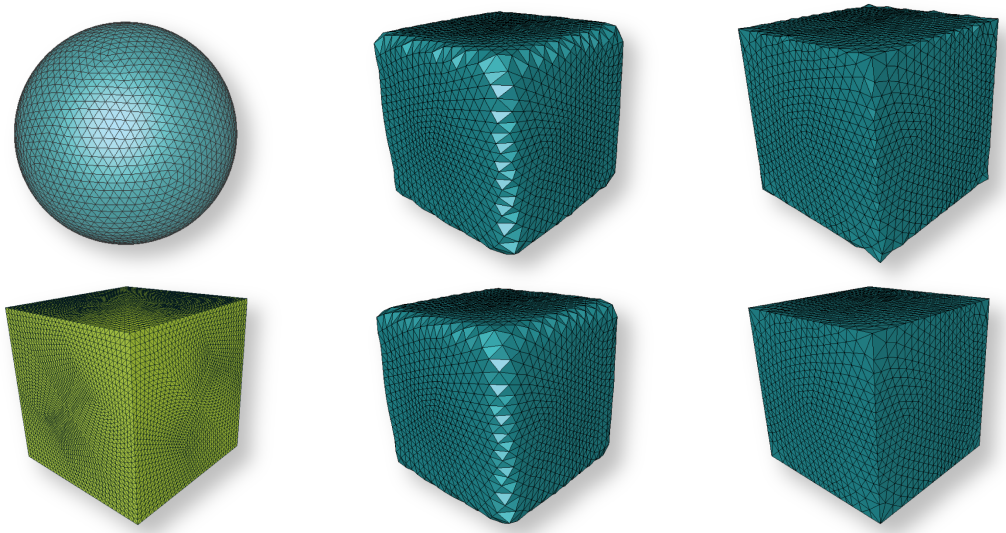

Figure 3.19. We optimize the sphere (5k triangles) on the left with respect to the green cube (18k triangles), which we use as the reference geometry. After just one iteration (middle column) of our algorithm the optimized meshes for 0 subdivision steps (top) and 4 steps (bottom) still look a bit bumpy, but the reference geometry is already clearly visible. After 10 iterations (right column) the mesh with 0 subdivision steps (top) still has not fully adapted to the reference geometry, while the vertices of the mesh with 4 steps (bottom) snap directly onto the feature lines of the cube.

sampling the interior of each triangle with more correspondence points (1 to 5 subdivision steps) penalizes such behavior and results in a much better approximation to the Hausdorff distance right from the start of the algorithm.

It has to be noted though, that since the linear system becomes less sparse, the runtime per iteration is slightly increased and as Figure 3.18 confirms the benefit of adding additional correspondence points becomes negligible after sufficiently many iterations.

Chapter 4

Shape Interpolation

In Chapter 3 we have seen, that it is possible to optimize triangle meshes such that they match certain specific criteria. Most of the presented applications dealt with static meshes, i.e. meshes that never changed their geometry over time. Yet, the use case mentioned in Section 3.3.1 is slightly different, because we applied our Mesh Massage framework to a *sequence* of meshes $\boldsymbol{M} = (M_1, \ldots, M_n)$ that captures the motion of a three dimensional object; the dancing man in this particular case.

Each mesh M_i reflects the geometry of that object but for a different point in time. If we pick a sample vertex $\boldsymbol{v}^{(1)} \in M_1$ from the first mesh of the sequence and gather its positions in every consecutive mesh $\boldsymbol{v}^{(i)} \in M_i$ we can create a piecewise linear curve $\boldsymbol{p} = (v^{(1)}, \ldots, v^{(n)})$, a *vertex path*. Seen from a different point of view, one could say that we used the framework to optimize the vertex paths for the animation sequence in Section 3.3.1 and thus reduced the wobbling effect of the texture in that application.

In the just mentioned example, the smoothness of the vertex paths was simply defined by the sheer number of given meshes in that sequence. In many cases, the purpose of an animation sequence is to create the illusion of motion for an object and obviously, the simulated motion gets smoother the more meshes we have. Unfortunately, more meshes also mean a larger amount of data. This raises an interesting question: Is it possible to create smooth and visually plausible vertex paths, from just a small set of input meshes?

The answer is immediately given by the amount of paper work that exists on mesh/shape interpolation. This chapter focuses on possible interpolation methods between two or more given input meshes and identifies the correct interpolation of rotations as one of the main difficulties.

The next section describes the application of deformation gradients [SP04; SZGP05; Sum06] to the task of interpolation. Although they are quite flexible and easy to implement, they are well known to produce counterintuitive results in the presence of large rotations. We will review their properties briefly and present several approaches, which we commenced to improve their behavior. Unfortunately, these approaches were

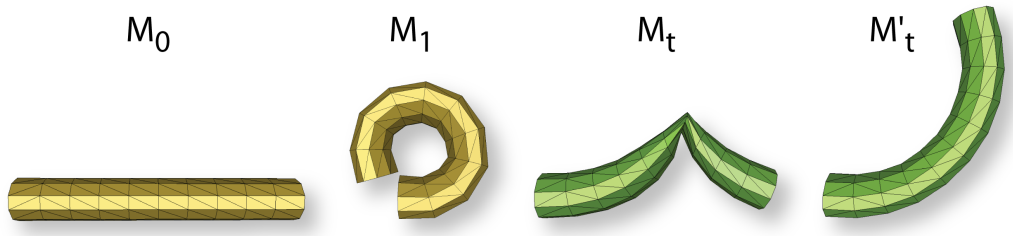

Figure 4.1. Interpolation between M_0 and M_1 yields M_t for $t = 0.3$, which is a rather non-intuitive result. Instead we would expect M_t' as the correct interpolation result.

not able to produce the desired results and we explain, why they were bound to fail. Finally, in Section 4.2 we present a method that is able to overcome the depicted challenges.

4.1 Deformation Gradients and Large Rotations

Although deformation gradients (see Appendix A) provide a great tool for modeling and interpolation between triangle meshes, sometimes they produce an unexpected result. This section explains why they behave unexpected in certain situations and describes several approaches that try to circumvent this behavior.

Consider Figure 4.1 for a second, as it is an example of what was just superficially described as "unexpected" behavior. If we interpolate between M_0 and M_1 using deformation gradients, a certain amount of triangles in the cylinder follows a wrong path through the interpolation [SP04; Sum06]. For example, for $t = 0.3$ the resulting mesh is M_t. Obviously, we would expect a result that looks more like M_t' in this case. In the following we shall see what the reason for this strange result is.

Deformation gradients model the transformation between a source and a target triangle with respect to a reference frame (see App. A, Fig. A.2, page 111). In case of the example in Figure 4.1, we have chosen M_0 as the reference mesh, thus the deformations for all triangles in M_1 are computed with respect to the triangles in M_0. To gain better insight, this setting is depicted on the left side of Figure 4.2. The right side reflects the equivalent situation in 2D with line segments s_i of unit-length. The straight (piecewise continuous) line L_0 should be transformed into L_1 by interpolating the coordinates of the points P_i in an appropriate way. From the interpolated line L_t we spot that the line segment s_{i+1} rotates the "wrong" way, i.e. it rotates *clockwise* instead of *counter clockwise*, while the previous segment s_i does not.

Since none of the other line segments influence the rotation, we can visually "lo-

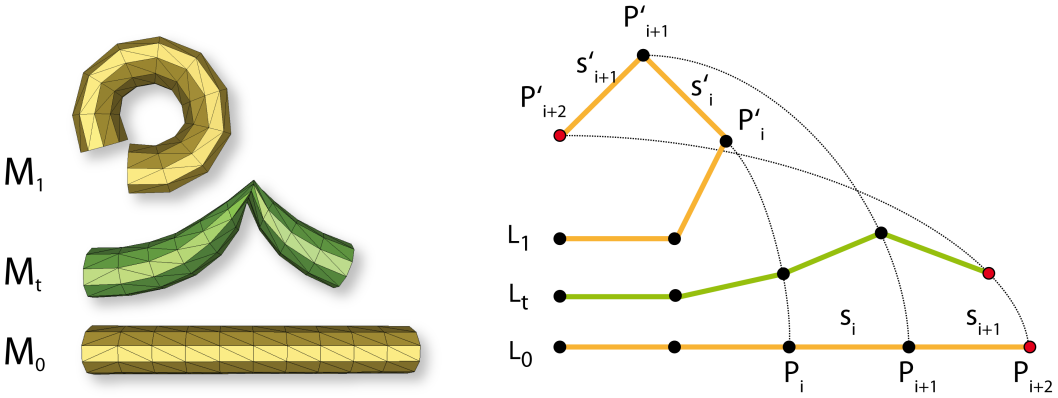

Figure 4.2. Interpolation in 3D (left) between M_0 and M_1 as well as in 2D (right) between L_0 and L_1. In 2D, one vertex adjacent to the line segments S_{i+1} follows a wrong path in the interpolation, causing an counter intuitive interpolation result. A similar ambiguity appears in the 3D setting.

calize" the situation further, by overlaying the corresponding line segments in a local coordinate system, as shown in Figure 4.3. In the left picture the rotation for the line segment s'_i is computed w.r.t. to the corresponding line segment s_i in the reference line, in this case L_0. The right side of the figure shows the case for the succeeding segment s'_{i+1} and its reference s_{i+1}. By defining the following function $R_2 : \mathbb{R} \to SO(2)$

$$R_2(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

we are able to express the rotation of the line segments as rotations of the points P_{i+1} and P_{i+2} into P'_{i+1} and P'_{i+2} as

$$\begin{aligned} P'_{i+1} &= R_2(\tfrac{3}{4}\pi) \cdot P_{i+1} \\ P'_{i+2} &= R_2(\tfrac{5}{4}\pi) \cdot P_{i+2}. \end{aligned}$$

with $P_i, P'_i \in \mathbb{R}^2$. It is rather obvious that we could have computed P'_{i+2} as well by a negative rotation, since

$$P'_{i+2} = R_2(\tfrac{5}{4}\pi) \cdot P_{i+2} = R_2(-\tfrac{3}{4}\pi) \cdot P_{i+2}.$$

This well known ambiguity is part of the aforementioned "misbehavior". As described in Appendix A (Sec. A.2.2, p. 113), the rotational part of the deformation gradients is converted into an axis-angle representation. Therefore, we extract the angle of rotation from the rotation matrix with the inverse function of the cosine. The arccosine,

$$\alpha = \arccos\left(\frac{\text{trace}(R) - 1}{2}\right)$$

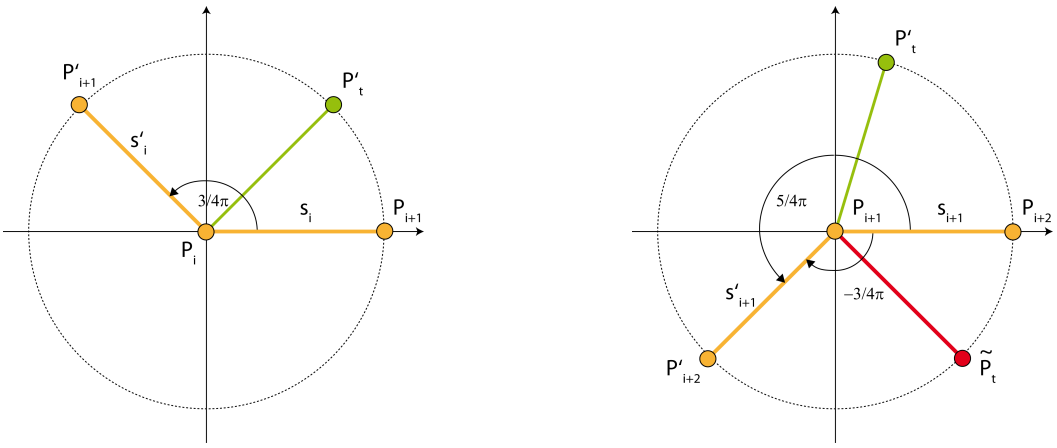

Figure 4.3. The rotation of two consecutive line segments, s_i to s_i' (left) and s_{i+1} to s_{i+1}' (right), sketched in a common local coordinate system. Interpolating P_{i+2} and P_{i+2}' yields $\tilde{P}_t$ instead of the expected P_t'.

which, by definition, returns values between 0 and π. By solely inspecting the arccosine of the angle, we are thus not able to identify correctly, whether we are rotating in the upper or in the lower hemisphere of the unit-circle. This is the reason the interpolation produces such strange results.

Assume that we want to interpolate between P_{i+2} and P_{i+2}' as shown in the right part of Figure 4.3. We know the result should be P_t', but instead we get $\tilde{P}_t$ due to the aforementioned situation. Fortunately, this can be fixed because we are currently not using all available information. A 2D rotation matrix can also be expressed as

$$R_2(\theta) = I \cdot \cos\theta + B \cdot \sin\theta$$

as mentioned in Section A.2.1, on page 112 in Eq. (A.3). Written in that form,

$$R_2(\tfrac{3}{4}\pi) = \tfrac{1}{2}\begin{pmatrix} -\sqrt{2} & -\sqrt{2} \\ \sqrt{2} & -\sqrt{2} \end{pmatrix} = I \cdot -\sqrt{2} + \underbrace{\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}}_{=: B_1} \cdot \sqrt{2} \tag{4.1}$$

$$R_2(-\tfrac{3}{4}\pi) = \tfrac{1}{2}\begin{pmatrix} -\sqrt{2} & \sqrt{2} \\ -\sqrt{2} & -\sqrt{2} \end{pmatrix} = I \cdot -\sqrt{2} + \underbrace{\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}}_{=: B_2} \cdot \sqrt{2} \tag{4.2}$$

we observe that additional information can be extracted from the matrices B_1 and B_2 of both expressions. For the lower hemisphere the entries of B_2 are inverted, compared to the entries of B_1 for the upper hemisphere. In two dimensions, we only have an

imaginary rotation axis, pointing in (left handed coordinate system) or out of (right handed system) the plane.

The situation in 3D only seems slightly more complicated, since the rotation axis can be arbitrarily oriented. In fact, the situation is the same, because the rotation also happens in the plane orthogonal to that axis and the axis extracted by the matrix logarithm is inverted in the same way. Let us quickly verify it, by defining $R_3 : \mathbb{R} \times \mathbb{R}^3 \rightarrow SO(3)$ as the matrix that is responsible for a rotation around the normalized vector $\boldsymbol{a} \in \mathbb{R}^3$ with a certain angle. Again, it is obvious that

$$R_3(\tfrac{5}{4}\pi, \boldsymbol{a}) = R_3(-\tfrac{3}{4}, \boldsymbol{a})$$

and from the Rodrigues Rotation Formula (Sec. A.2.2, p. 113))

$$I + B \cdot \sin\theta + B^2(1 - \cos\theta)$$

we see that for $\theta > \pi$ the last term of the equation stays positive, but $B \cdot \sin\theta$ becomes $-B \cdot \sin\theta$. Similar as in 2D, the matrix logarithm in 3D encodes the rotation of $\frac{5}{4}\pi$ around $\boldsymbol{a}$ as

$$R_3(\tfrac{5}{4}\pi, \boldsymbol{a}) = R_3(\tfrac{3}{4}, -\boldsymbol{a}).$$

In other words, a rotation of $\frac{3}{4}\pi$ around the *inverted* axis $-\boldsymbol{a}$. We can put this knowledge to use in the same way as in the 2D case and check whether we are rotating in the upper or lower hemisphere. For example, given the rotation matrices $R_3(\frac{3}{4}\pi, \boldsymbol{a})$ and $R_3(-\frac{3}{4}\pi, \boldsymbol{a})$, we cannot decide – by purely inspecting the rotation angle – in which hemisphere we are currently working. But, if we extract the rotation axis with Eq. (A.7) (Sec. A.2.2, p. 113) we see that a negative angle is represented with an inverted axis in the matrix logarithm,

$$\begin{aligned} R_3(\tfrac{3}{4}\pi, \boldsymbol{a}) - R_3(\tfrac{3}{4}\pi, \boldsymbol{a})^T &= \begin{pmatrix} 0 & -1 & 1 \\ 1 & 0 & -1 \\ -1 & 1 & 0 \end{pmatrix} \cdot \frac{1}{2\sin\theta} \\ R_3(-\tfrac{3}{4}\pi, \boldsymbol{a}) - R_3(-\tfrac{3}{4}\pi, \boldsymbol{a})^T &= \begin{pmatrix} 0 & 1 & -1 \\ -1 & 0 & 1 \\ 1 & -1 & 0 \end{pmatrix} \cdot \frac{1}{2\sin\theta}. \end{aligned}$$

This leads to the conclusion that the correct interpolation of the rotation angles heavily depends on the orientation of the reference line/triangle, that was used for computation of the deformation gradients.

Baran et al. [BVGP09] for example, exploit this fact, by orienting the reference triangles such that the rotation interpolation always produces proper results for a certain cluster of triangles. Unfortunately, for extremely large rotations (i.e. larger than π w.r.t. the reference triangle) the exposed problem tends to reoccur in general.

Sumner et al. [SZGP05; Sum06] solve the problem of ambiguous rotations in the most intuitive way. They let the user decide. Triangles are selected manually, then a

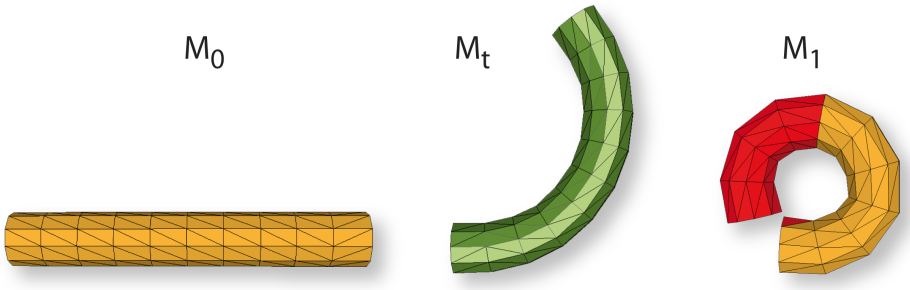

Figure 4.4. The modified interpolation between M_0 and M_1 yields M_t for $t = 0.5$. Without modification the red triangles would rotate clockwise instead of counter-clockwise (see Fig. 4.2). The modified approach uses a Breadth-first-search to prescribe a rotation path, which allows to detect and correct those triangles in M_1.

corrective multiple of 2π is added. For an ordinary model like the cylinder in Figure 4.2 it is maybe appropriate to allow for such manual corrections.

Yet, finding and manually selecting the triangles which should be corrected, is quite an exhausting task for reasonably sized meshes. The results from the previous section raised the question, if it is possible to find a path in the mesh, such that following this path and propagating the rotation angle from one triangle to the next, leads to the correct overall rotation. For example, if you take a look at the right picture of Figure 4.2, it is evident to us, in which direction the last line segment should rotate and we could cast our knowledge into an algorithm in the following way.

Starting from the leftmost line segment, we trace the angle of rotation for every following line segment. In combination with the results from the previous section this allows us to distinguish between the possible rotations and correct them if necessary. Assuming a smooth deformation from M_0 into M_1, we are capable of detecting a "wrong" rotation by inspecting the signs of the of the matrix logarithm B (see Eq. (4.1) and Eq. (4.2)).

In case of Figure 4.2 we detect a continuously increasing rotation angle, until we reach the last line segment. At this point the difference between the previous rotation angle and the current one, is large and we find that the matrix B of the current segment is inverted compared to the matrix of the previous segment. In this case, the rotation angle is updated to $\theta' = 2\pi - \theta$ and the matrix B is inverted.

This approach works well in 2D, since all rotations happen in the same plane or in other words around the same axis. However, in 3D the solution to the problem is not that obvious, due to the additional degree of freedom the arbitrary oriented rotation axis provides. The following sections address several approaches to compute a *rotation path*, but also their shortcomings.

4.1.1 Breadth-First-Search Traversal (BFS)

Our first approach in 3D is a straightforward implementation of the idea sketched in the previous section. The initialization phase computes a consistent traversal order for all input meshes, based on a breadth-first search (BFS) that starts from a random triangle. This pre-computed traversal order provides us with a rotation path, relating two consecutive matrix logarithms to each other.

Along this path the dot product between the rotation axes of two consecutive triangles is computed. In relation to the result of the previous computation, we check if the sign of the dot product is inverted and whether the difference in the rotation angle is large. If both conditions are fulfilled, we found a triangle whose matrix logarithm needs to be "corrected". Therefore, we flip the current rotation axis by inverting the matrix B and update the rotation angle according to the precedent triangle. The following algorithm should clarify the approach.

BFS-Algorithm

1. compute the BF traversal order, starting from a random triangle
2. for each input mesh do
   (a) compute rotation angle θ_0 and rotation axis a_0 for the first triangle
   (b) proceed with the next triangle that has index k and compute the angle θ_k and axis a_k
   (c) compute the dot product $s = \langle a_k, a_{k-1} \rangle$
   (d) if $s \geq 0$ proceed with (2b), otherwise go to (2e)
   (e) invert the rotation angle and the axis, $a_k = -a_k$, $\theta_k = -\theta_k$
   (f) the corrective multiple of 2π is computed as $q = \lfloor \frac{\theta_{k-1} - \theta_k}{2\pi} \rfloor$, $q \in \mathbb{Z}$
   (g) update the rotation angle to $\theta_k = \theta_k + 2q\pi$
   (h) while there are triangles in the path go to (2b)

Figure 4.4 confirms that this approach works quite well for the cylinder model, since M_t rotates into the anticipated direction. Again, M_0 (left) was used as one of the input meshes and at the same time as the reference mesh. The color coding of M_1 visualizes the triangles, for which a flip in the rotation axis has been detected by the BFS algorithm.

Encouraged by the results of the cylinder model, we were eager to see, if the algorithm performs equally well for models that are more complex. The famous armadillo model (see Figure 4.5), obviously suffers from the matrix logarithm problem as well, induced by triangles in the left knee and right foot of the model.

Figure 4.5. Interpolating the armadillo without modifications. M_t (middle) for $t = 0.5$ clearly reveals that some triangles would rotate the wrong way from M_0 to M_1, causing the armadillo's left knee and right foot (compare Figure 4.6) to fold.

These problematic (red) triangles are detected and modified by the BFS algorithm as Figure 4.6 shows, the interpolation result M_t for $t = 0.5$ also looks like expected. The setup in this case was the same as for the cylinder model, M_1 provided the target pose and M_0 was used as source pose, as well as the reference mesh for the deformation gradients.

Although the BFS traversal algorithm performs quite well for these models, we need to act with caution, due to its simplicity. For example, the rotation of a triangle with respect to its reference triangle could be exactly a multiple of 2π. In this case the rotation matrix is equivalent to the identity matrix and the change in the rotation angle would be zero. This would obviously prevent the propagation of the correct rotation angle. We select the correction factor q, in such a manner that the difference between the current rotation angle and the previous one is minimal, to circumvent this behavior. We have two possible choices for q here, one for positive and one for negative angles. Taking this fact into account, we compute q as

$$q = \lfloor \frac{\theta_{k-1} + \text{sign}(\theta_{k-1})\pi}{2\pi} \rfloor.$$

Another problem arises if the rotation inherent in the deformation gradient is very small, remember that we used algorithm (A.2) (Sec. A.1, p. 112) to separate the deformation gradient into a rotation R and a scale/shear S part. It is possible, that due to rounding errors in the polar decomposition, a rotation axis may be identified as a "false-positive" by the algorithm. The rounding errors originate either from the iterative nature of the algorithm or simply from the lack of machine precision. We prevent the detection of these "false-positives" in the following way.

The decision, whether the axis and the angle should be modified, depends on the difference in the angle as well as on the sign of the dot product of the rotation axes

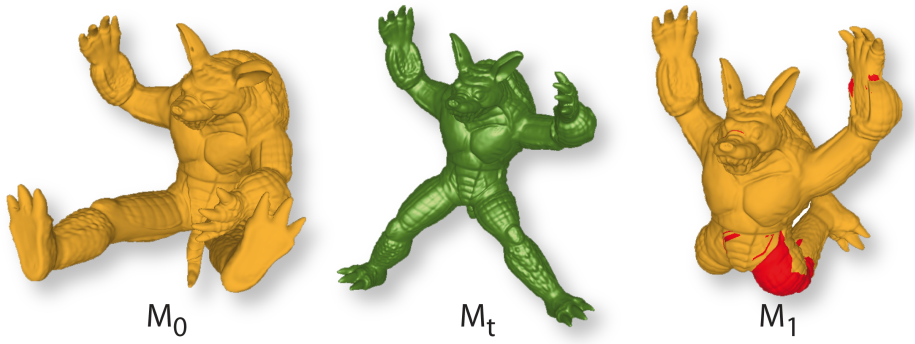

Figure 4.6. Interpolating the armadillo using the BF traversal algorithm. The red triangles in the left knee of the armadillo in the target pose M_1 are identified by the algorithm and their matrix logarithms are rectified, leading to a better result M_t (middle) for $t = 0.5$.

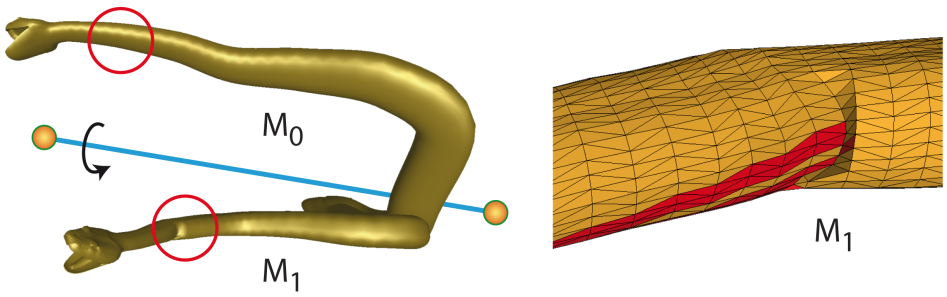

Figure 4.7. Limitations of a simple BF search traversal. On the left the source (M_0) and the target pose (M_1) are shown. While the surface of M_0 is smooth, a crease near the snake's head is visible in M_1. Due to this crease, the red triangles in the magnification on the right side, are spuriously detected and cause a wrong rotation axis to be propagated.

between the current triangle and its predecessor in the path. Hence, to compensate for the rounding errors, we check if the difference is less than a certain threshold (one degree in our implementation) and then simply skip the triangle, proceed with its predecessor and compute again.

Yet, the biggest disadvantage of the algorithm stems from the fact that our assumption of a smooth transition in the rotation angle throughout the traversal path is not necessarily true for arbitrary meshes. Consequently, the simple BFS traversal algorithm spuriously detects a flipping of the rotation axis which then gets propagated. Figure 4.7 gives an example of such a situation. The left side of the picture shows two meshes

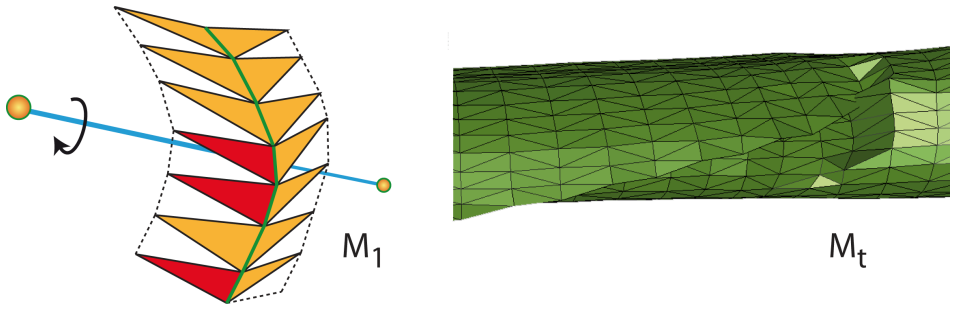

Figure 4.8. An illustration of the situation around the crease in M_1. The red triangles cause a false propagation of the rotation axis, which leads to a highly disturbed surface in the interpolation result M_t.

capturing different poses of the snake model. The upper one (M_0) serves as the source pose and reference mesh, the lower one (M_1) is the desired target pose. Noteworthy in this case is the fact that M_0 has a completely smooth surface, while there is a crease in M_1. This small discrepancy has big influence on the interpolation result M_t, as shown on the right of Figure 4.8. The surface of the model is amply deformed, due to the red triangles wrongly detected in M_1.

The situation is depicted in the left of Figure 4.8 to clarify why false-positive triangles are detected. First of all, the snake is traversed from right to left in this case. Secondly, beside the pictured crease, there is almost no deformation present in the surface of the model. In other words, the majority of triangles that are not part of the crease are rotating around the “global” rotation axis sketched in blue. While traversing the calculated rotation path, the BFS algorithm checks the dot product of the rotation axes of consecutive triangle pairs, as we have seen above. For triangles on the right side of the defect, this works pretty well, since they are only subject to the global rotation. The same is true for the triangles on the left side. On the contrary, triangles which are part of the crease have to rotate around a second “local” axis as well (depicted in green, which is the common edge between two adjacent triangles), to accommodate for the different geometry.

If the algorithm now checks the dot product between a triangle that is mainly influenced by the global rotation and a triangle that is part of the crease, the rotation axes are almost orthogonal, which is depicted in the left of Figure 4.8. Depending on the orientation of the global rotation axis and the local ones, the algorithm falsely detects a flip for some triangles, and this flip gets then erroneously propagated. To prevent this propagation we extended the algorithm as described in the next section.

4.1.2 Prioritized Matrix Logarithm Traversal (PML)

The aforementioned problem is caused by the BFS algorithm itself. Due to the fact, that the traversal order is not coupled in any way to the mesh geometry. Thus, we modified our first approach to such an extent, that we utilize a priority queue for calculating the traversal order.

At the beginning of this chapter we saw that the axis angle representation of the rotational part of the deformation gradient, the matrix logarithm, is a combination of the normalized rotation axis and the rotation angle. In other words a vector in $\mathbb{R}^3$. Therefore, we modify the BFS traversal algorithm such that it selects that neighboring triangle, which offers the smallest squared distance in the matrix logarithm and has not been visited yet. This triangle is then added as the next triangle to the path. The following pseudocode clarifies this approach

PML-Algorithm

1. for each input mesh do
   (a) calculate the "uncorrected" matrix logarithm for each triangle
   (b) setup a priority queue and add the start triangle
   (c) take the first triangle Δ from the queue (which is automatically the one with the smallest difference in the log compared to the predecessor)
   (d) for all neighbors Ω which are not visited yet, compute their priorities P, $p_\omega = ||\log_\Delta - \log_\omega||_2^2 \quad (p \in P, \omega \in \Omega)$
   (e) push all $\omega \in \Omega$ with their priority p_ω into the priority queue
   (f) if necessary, correct angle and axis of triangle Δ
   (g) go to step (1c) until all triangles are visited

In case of the cylinder, the algorithm performs similar to the simple BFS approach, because it also allows to capture the smooth change in the geometry between M_0 and M_1 properly. Yet, for the snake model this small modification also prevents the propagation of the wrongly detected rotation axes, as shown on the left (M_1) of Figure 4.9. Although there are still triangles in the crease, which are detected spuriously, they cannot propagate their wrong axis any longer, due to the changed traversal order. The priority queue takes care that triangles connected to the crease are visited last, which considerably improves the interpolation result M_t compared to the one in Figure 4.8.

Although the priority queue improved the robustness in case of the snake model, we have to be careful to avoid another pitfall. The traversal order and thus the outcome of the interpolation still heavily depends on the orientation of the first triangle in our path. For example, if the rotation angle from the reference triangle to the corresponding triangle in the input mesh is already close to π, then a small change in the

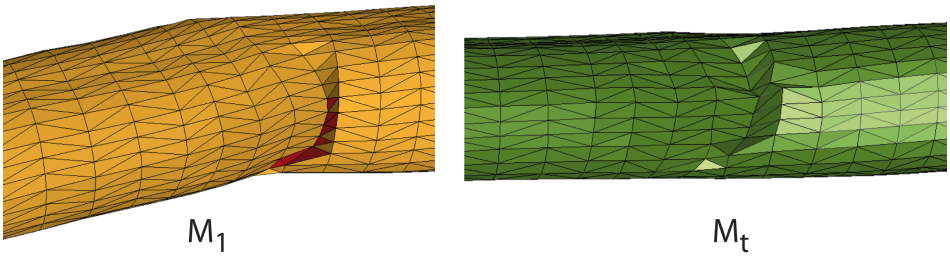

Figure 4.9. A modified traversal algorithm (PML), which heeds the crease. Although some triangles (red) are still falsely detected, their influence on the rotation path is minimized, with the help of a priority queue.

matrix logarithm could lead to the detection of a flipped rotation axis, although this "movement" is due to the global rotation and not caused by the local deformation that we rather would like to capture. This is illustrated in Figure 4.10 where you can see that the cylinder on the left was subject to a global rotation. If we use the undeformed cylinder as reference mesh, the deformation gradients capture two major rotation directions, r_2 that is responsible for the "coiling" and r_1 that aligns both cylinders. But, in terms of interpolation we are only interested in capturing the rotation that causes the "coiling". The armadillo in Figure 4.10 shows the behavior of a reasonably sized model in the presence of a global rotation.

Of course, for a *simple* model, this can easily be fixed with a global alignment step prior to the computation of the deformation gradients and the following mesh traversal as shown on the left of Figure 4.10. The term *simple* in this case does not refer to an elementary model, but to an elementary rotation of the model.

A good example thereof is the cylinder: it bends smoothly from the target to the source pose following one main direction. From this point of view, the armadillo is also simple, because the arms and legs undergo a similar rotation in the same direction. Hence, aligning the start triangles improves the results in case of the cylinder and the armadillo.

Nevertheless, for a mesh that incorporates many local rotations around varying rotation axes, aligning the start triangles is no longer sufficient. This is emphasized by the lion model in Figure 4.11. For example, there is one rotation which is responsible for the convolving effect, another one that causes the target pose to rest on the imaginary floor and finally several rotations around different axes are responsible for the movement of the tail and the paws.

Due to the fact that all these different rotation axes originate from the computation of the deformation gradients with respect to their *reference* triangle, it is rather difficult to decide for each triangle whether we should use it for tracing the correct rotation. We

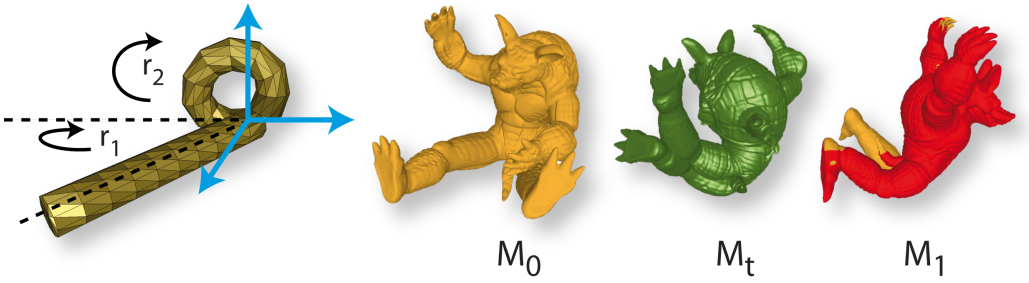

Figure 4.10. If more than one rotation is present in the input models, as shown on the left of the picture, the PML traversal algorithm is bound to fail. In case of the cylinder we would like to capture the rotation r_2, but also have to account for the global rotation r_1. The same holds for the shown armadillo, where we would like to capture the rotation of the arms and legs, but also have to account for the global rotation between M_0 and M_1 as well. Although, this global rotation can be factored out easily in the above cases, unfortunately this is not always possible (compare Fig. 4.11).

cannot explicitly specify the correct (expected) rotation any longer. Hence, we modified the traversal algorithm once more, such that it now takes the normal information of the triangles into account. The next section details the approach.

4.1.3 Prioritized Matrix Logarithm with Normal Difference (PMLN)

The major drawback of the PLM algorithm is that it does not capture the triangle's deformation with respect to its surrounding neighbors, but with respect to some reference frame which is not necessarily related. Thus, it may occur that although the difference in the matrix logarithm to the reference frame is small, we are dealing with a highly varying curvature at that part of the mesh.

For example, this could be a local self intersection in one of the meshes which was also used as reference mesh in this case. Therefore, we modified the PML algorithm further by taking the angle difference between two consecutive triangle normals into account, leading to the PMLN algorithm. We set the priority queue's sorting criterion such that for two consecutive triangles the deviation in the normals is combined with the difference in the matrix logarithms. The smaller the differences, the higher the priority of the corresponding triangle.

If we define the sorting criterion in this way, then we favor areas of the mesh that are rotating similarly (matrix log) and where the change in curvature is small (normal deviation). Concurrently, we postpone the processing of areas where we are not certain what the correct interpolation behavior should be. In this setting we assume that a rotation path exists for the mesh, if the mesh's surface is mainly smooth. Considering

Figure 4.11. The prioritized matrix log traversal applied to the lion model performs not as expected. The green meshes visualize the interpolation results for $t = 0.25, t = 0.50, t = 0.75$. The meshes to the left and right provide the input poses.

the snake model, the PMLN algorithm performs equally well compared to the PML algorithm. Similarly, it cannot avoid the spurious detection of flipped axes, but it can prevent their expansion, by inserting the problematic triangles at the end of the priority queue. Yet, for the lion model shown in Figure 4.12 the algorithm is able to find a rotation path that results in a plausible interpolation, despite the many difficulties present in the model.

The figure also summarizes the results of the different traversal algorithms presented in this chapter. While the leftmost column reflects the source and reference pose, the rightmost column shows the target model in the lower row and additionally a flat shaded model (top) that visualizes the triangles for which a flip in the rotation axes was detected by the corresponding traversal algorithm. For better comparison, the flat shaded model is rotated slightly. The snapshots in the middle correspond to the interpolation results for $t = 0.25, t = 0.50$ and $t = 0.75$. The top row shows the results of the BFS traversal algorithm. Due to the fact that this algorithm does not respect the mesh geometry and its deformation throughout the interpolation, it performs not very well for this model.

The middle row presents the result for the PML algorithm and the accentuated target mesh emphasizes that the algorithm is not able to correctly detect the local rotations for this mesh. Only a few triangles in the right paw and the tip of the tail are detected as flipped. Hence, the interpolation results differ only slightly from the direct interpolation without any correction at all. This is, as already mentioned above, caused by the existence of multiple different rotation axes in the mesh.

The lower row finally displays the results for the PMLN algorithm. The in-betweens show that this algorithm provides by far the best results compared to the other algorithms. Using a priority queue with this sorting criterion forces the traversal algorithm to find a path in the mesh that changes smoothly and allows us to captured the desired

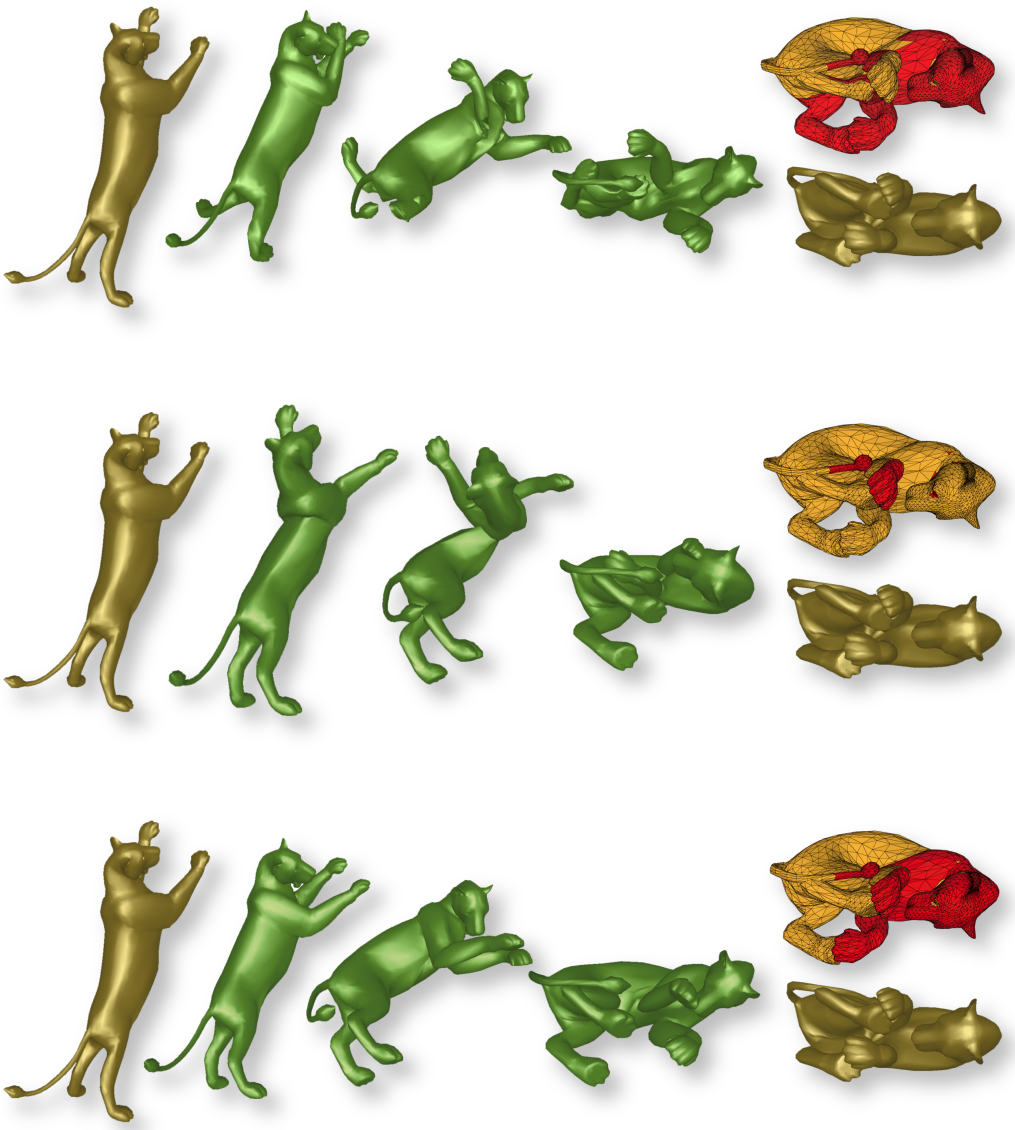

Figure 4.12. Comparison of different traversal methods applied to the lion model. From top to bottom the results of the BFS (Sec. 4.1.1), PLM (Sec. 4.1.2) and PLMN algorithm (Sec. 4.1.3) are shown. The left and right most column reflect the input meshes. The green meshes in between visualize the interpolation results for $t = 0.25, t = 0.50, t = 0.75$ for the corresponding method.

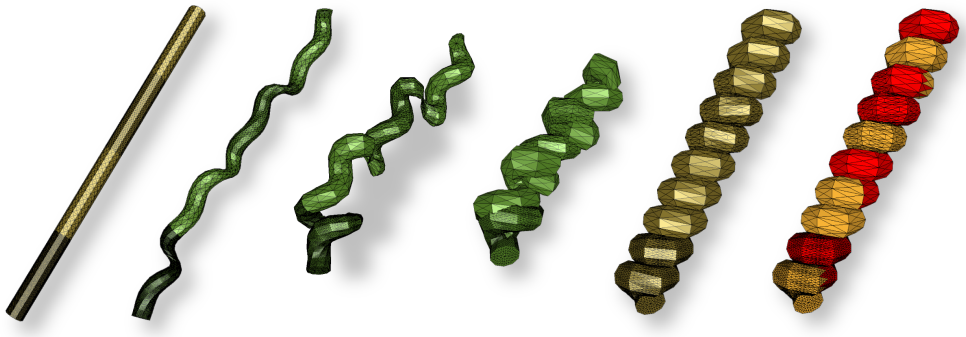

Figure 4.13. Although the PMLN algorithm yields acceptable results for the lion model, the helix model underlines, that it cannot be guaranteed to find a suitable rotation path for arbitrary models. The models depicted in gold are the input models, the green ones reflect the interpolation results for $t = 0.25$, $t = 0.5$, and $t = 0.75$. The rightmost model shows the falsely detected triangle flips, caused by the inappropriate traversal path.

rotation, but the main problem is not solved, we are still computing the deformation with respect to some reference frame.

One could argue that the PLMN algorithm works very well, which is true for the lion model, but the problem of the different local rotations remains. This becomes rather obvious, if we inspect the behavior of the helix model in Figure 4.13. The setting is the same as for all the other examples. The source pose, which provides also the reference mesh, is shown on the left side and the second model from the right reflects the target pose. The rightmost column shows the target pose again, but this time the triangles are color coded to emphasize the result of the PMLN algorithm. The algorithm starts from the bottom of the cylinder and ideally should track all the rotations in the helix, but obviously fails to do so as confirmed by the bad interpolation results depicted in green. It should become obvious in the next section, why tracing rotation paths in deformation gradient interpolation is not a good choice.

4.1.4 Drawbacks

Although deformation gradients offer a valuable tool for many applications in computer graphics, they reveal certain inadequateness if used for geometry interpolation purposes. More precisely, if the input poses differ in large rotations (i.e. larger than 180 degrees), deformation gradients are by construction not able to provide a visually plausible direct interpolation result. As shown at the beginning of this chapter, the creation of a rotation path for tracing and fixing wrong rotations is possible in the 2D

setting of the problem. Unfortunately, using the same approach in 3D is bound to fail, since it is not obvious at all how to build a rotation path and we cannot even be sure if it exists for an arbitrary transforming/morphing mesh.

The degrees of freedom that deformation gradients offer, are sufficient to *express* an arbitrary orientation of a triangle in 3D and thus they are well suited for applications such as deformation transfer from one mesh to another one, but their *direct interpolation* using the matrix logarithm does not yield the desired result.

The representation of their rotation component as a combination of an axis and angle enables easy blending between two or more deformation gradients, but at the same time this easy blending restricts the interpolated rotation axis. Since we can only subject it to follow a great circle on the unit sphere (see App. A.2.3) and this is not necessarily the correct path the triangle should follow to create plausible interpolation result. This is well emphasized by the lion and the helix examples at the end of the previous section.

The assumption that a spherical linear interpolation (Slerp) between the source rotation axis and the target rotation axis reveals the correct transformation of the triangle does not have to be true for an arbitrary transformed mesh. For example, given a source and a target triangle, the interpolated triangle has the possibility to rotate around an axis orthogonal to the source and target normal. Additionally, it can rotate in the plane orthogonal to its normal.

Deformation gradients can represent any orientation that can be built from these two possible rotation directions, but since this combination of rotations is then expressed by a single rotation axis, the path of the axis on the unit sphere very seldom conforms to a spherical linear interpolation between the source and target axis.

If you take a look at Figure 4.14 it is now obvious why it is possible to trace, fix and then correctly interpolate the cylinder model. The left picture shows the path (green) of a triangle's normal throughout the interpolation. The red curve is the *Slerp* (see Sec. A.2.3) between the start and target normal for the very same triangle. The triangle was picked randomly, but with the constraint that it does not rotate more than π, in other words it was sampled from the left side of the cylinder. On the contrary, the picture in the middle and the one on the right side, reflect the behavior of a normal that was sampled from a triangle on the right side of the cylinder. The green line in the middle shows the normal path while interpolating with deformation gradients, while the red line again shows the *Slerp* between the start and target normal. The picture on the right shows the path for the same normal, but with correction. The deviation in the left and right picture is due to the final gluing step of the deformation gradient interpolation method (see A.3).

For the "curling" cylinder model, deformation gradients are able to capture the movement and allow for its interpolation. But if we switch to a slightly more complex model, such as the lion shown in Figure 4.15, the behavior of the rotation axis and

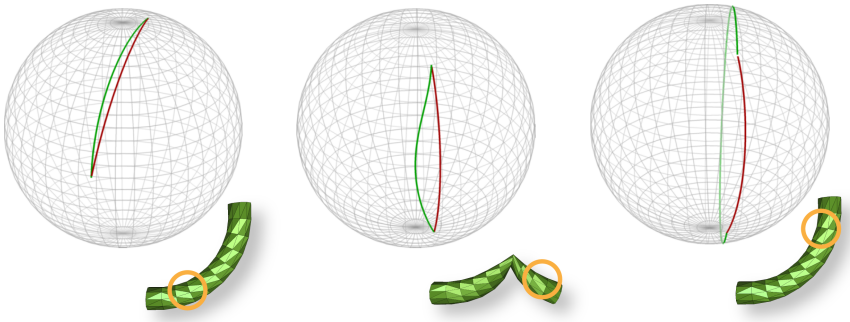

Figure 4.14. The green curve reflects the path of a randomly picked triangle's normal throughout the interpolation. The red curve is the *Slerp* (see A.2.3) of that normal. The yellow circle depicts the area where we sampled the triangle from. The left picture shows the paths for a triangle that is not rotating more than π. On the contrary, if the triangle is supposed to do so, but is not "corrected", we get the result shown in the middle. The right picture shows both paths for the same triangle as in the middle, but "fixed".

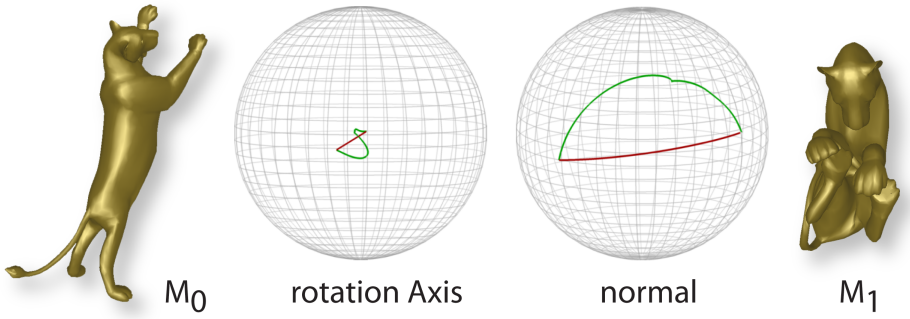

Figure 4.15. For a randomly chosen triangle the path of the rotation axis as well as the normal path is plotted. The green curve is sampled directly from the triangle after using the MSGI method (Sec. 4), the red curve represents the *Slerp'*ed version.

the normal for a randomly chosen triangle is significantly different. For these plots we applied a different interpolation method that is described in detail in Section 4.2. We picked a random triangle and sampled the normal for every interpolation parameter. The rotation axis was sampled similarly. First, we computed the interpolation with the MSGI method, then – for the triangle in question – we computed the deformation gradient and extracted its rotation axis.

Due to the many different local rotations present in this model, the path of the rotation axis, as well as the normal's path, that lead to a visually plausible result, are very far from their *Slerp*'ed counterpart that we are able to create with deformation gradi-

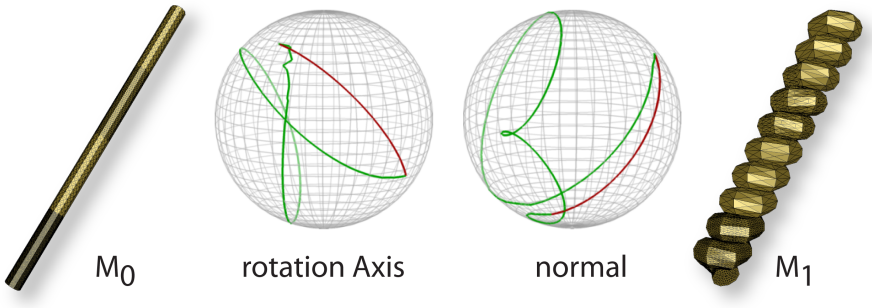

Figure 4.16. The same setup as for Fig. 4.15, but for the highly curled helix model. Again the curves are sampled from the triangle after applying the MSGI method. It is rather obvious that for a visually plausible interpolation the path is much different to what a Slerp'ed rotation axis with deformation gradients (including correction) is able to reproduce.

ent interpolation. This disadvantage becomes even more pronounced if we inspect the behavior of one of the triangles in the helix model in Figure 4.16.

The setting is the same as for all the other examples. The source pose, which provides also the reference mesh, is shown on the left side and the second model from the right gives the target pose. The normal path was as well as the rotation axis' path sampled as mentioned previously. It is rather obvious that the proper paths are by far not comparable to the Slerp'ed paths. Therefore, we cannot expect that interpolating with deformation gradients, even modified with a traversal path, will produce a visually pleasing interpolation result in the presence of large rotations.

Consequently, our contribution to the field of mesh interpolation is a very simple and fast method that interpolates triangles at the core of the method in a locally as-rigid-as-possible manner. At the level of adjacent triangles the problem of large rotations does not exist. The problem arises when more triangles should be interpolated, therefore we apply a global multi-registration procedure which allows us to track the global rotations. Finally, we speed up the approach by using a hierarchal structure of mesh patches.

4.2 Multi-Scale Geometry Interpolation

Our method is based on a hierarchical approach and is similar in spirit to the shape matching in [MHTG05; BPGK06; SA07]. At the bottom of the hierarchy, we consider single triangles and linearly interpolate the local metric of the given meshes. Let A_0 and A_1 be two corresponding triangles in the source and the target mesh with edge lengths

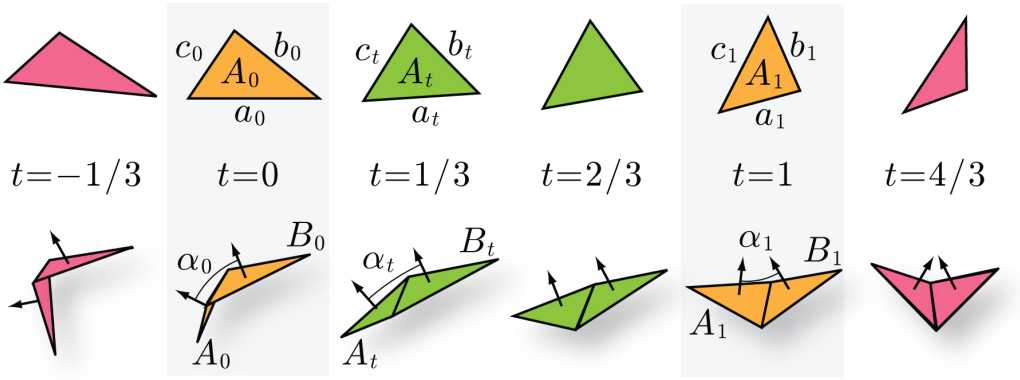

Figure 4.17. Linear interpolation of a single triangle (top) and a wedge (bottom).

a_0, b_0, c_0 and a_1, b_1, c_1. Given some interpolation parameter $t \in [0,1]$, we construct the destination triangle A_t by linearly interpolating the edge lengths (see Figure 4.17). That is, A_t is a triangle with edge lengths $a_t = (1-t)a_0 + ta_1$, $b_t = (1-t)b_0 + tb_1$, $c_t = (1-t)c_0 + tc_1$, and it is clear by construction that such a triangle always exists. However, although this determines shape and size of A_t, we do not yet know where to place it, and that is where the hierarchy comes into play.

On the next coarser hierarchy level, we consider pairs of adjacent triangles (wedges) and linearly interpolate the local mean curvature of the given meshes. Let (A_0, B_0) and (A_1, B_1) be such corresponding wedges in the source and the target mesh with dihedral angles α_0 and α_1 at the common edge (see Figure 4.17). We then glue the interpolated triangles A_t and B_t together such that they form a wedge with dihedral angle $\alpha_t = (1-t)\alpha_0 + t\alpha_1$. Note that A_t and B_t fit together seamlessly as the common edge has the same length in both triangles. Again, this does not tell us where to place the wedge (A_t, B_t) in space, but at least it determines the *relative position* of one triangle to the other.

In some sense, this method interpolates between corresponding wedges as rigidly as possible, because both the local metric (edge lengths) and the local mean curvature (dihedral angle) are interpolated in the straightest way (linearly). It now remains to paste all interpolated wedges together in order to yield the interpolated mesh. If we consider all possible wedges, that is, one for each edge in the mesh, then it is clear that for each triangle there are three wedges that overlap, like the scales of a fish. And it is this small overlap that can be exploited in order to arrange all wedges globally. All we have to do is to find a set of rigid transformations, one for each wedge, such that the overall sum of distances (or rather squared distances) between all corresponding vertices for two overlapping wedges is minimized. Such a global alignment procedure for all wedges can in principle be solved by a multi-registration method, but for meshes with more than a few hundred triangles this can become very slow and unstable.

Thus, we take further advantage of our hierarchical approach. On the next coarser level above the wedges in our hierarchy, we cluster all the wedges around a common vertex and align them with the multi-registration method of Williams and Bennamoun [WB00] (see Section 4.2.1). Once aligned, we average the coordinates of corresponding vertices (see Section 4.2.2) and combine the wedges to form a one-ring. If we do so for all vertices of the mesh, we get a set of larger patches, which again overlap by an even bigger amount (neighbouring one-rings share two triangles), like the scales of an armadillo. Proceeding this way recursively, we create larger and larger patches by always clustering, aligning, and blending a small number (4 to 10) of overlapping and neighbouring patches. And at the top of the hierarchy we get the interpolated mesh, similar to the way the scales of a crocodile form its exoskeleton.

Note that this method clearly reproduces T_0 and T_1 for $t = 0$ and $t = 1$, respectively, because in both cases the patches can be aligned with zero distance on all levels of the hierarchy.

4.2.1 Aligning Patches

Let us now take a closer look at how to align multiple patches, so that they form a globally consistent mesh. This problem is somewhat similar to the registration of point clouds in 3D. But our setting is even simpler, in that we already know all corresponding vertices and do not have to bother searching for closest points. Consider, for example, the alignment of two neighbouring wedges. Since they have an overlap of one triangle, there are exactly three corresponding vertex pairs (the corners of said triangle), and that suffices to align them optimally. And on all upper levels of the hierarchy, neighbouring patches overlap by even more triangles.

In order to register two neighbouring patches, we could apply the method of Besl and McKay [BM92] or one of its improved siblings and directly compute the optimal rotation and translation for transforming one patch such that it aligns best to the other. But unfortunately, pairwise registration leads to error accumulation in the hierarchy. We therefore need a method that allows to distribute the registration error between the patches as equally as possible.

We tested several such multi-registration methods [SH96; CS99; PLH02] and found the one of Williams and Bennamoun [WB00] to be very well adapted to our particular problem. Their goal is to *simultaneously* determine *all* rigid transformations (translation and rotation), i.e., one for each of the patches that need to be registered. This is done by minimizing a cost function which sums up all the squared distances between corresponding vertices for neighbouring patches.

More precisely, Williams and Bennamoun consider M overlapping "views" (our patches). Between these views, P sets of pairwise correspondences exit. The mappings $\alpha(\mu)$ and $\beta(\mu)$ for $\mu = 1, \ldots, P$ define the two patches which participate in the μth correspondence and N_μ specifies the number of corresponding pairs of points in each set. Points in the correspondence set μ which start in view $\alpha(\mu)$ are denoted by $\boldsymbol{x}_i^\mu$,

while points from $\beta(\mu)$ are denoted by $\boldsymbol{y}_i^\mu$. The goal is to find a transformation for each view, that minimizes the distance between $\boldsymbol{x}_i^\mu$ and $\boldsymbol{y}_i^\mu$. These transformations are expressed by a 3×3 rotation matrix R^m and a 3×1 translation vector T^m for $m = 1, \ldots, M$. Furthermore, we concatenate these per patch rotations into the block parameter matrix

$$R = \begin{pmatrix} R^1 & \cdots & R^M \end{pmatrix}$$

and in a similar way we collect the per patch translations in a large translation vector

$$T = \begin{pmatrix} T^1 \\ \vdots \\ T^M \end{pmatrix}.$$

This allows us to express the registration procedure as the minimization of a cost function Φ which measures the squared Euclidian distances between the transformed corresponding points over all correspondence sets:

$$\Phi(R,T) = \sum_{\mu=1}^{P} \sum_{i=1}^{N_\mu} \left\| \left(R^{\alpha(\mu)} x_i^\mu + T^{\alpha(\mu)} \right) - \left(R^{\beta(\mu)} y_i^\mu + T^{\beta(\mu)} \right) \right\|^2.$$

The key idea of Williams and Bennamoun's approach is to pre-compute in an elegant way a constant matrix that encodes the whole registration problem. Hence, they introduce the $3M \times 3P$ block selection matrices C^α and C^β defined by

$$\begin{aligned} C^\alpha_{m,\mu} &= \begin{cases} I_3, & \text{if } \alpha(\mu) = m, \\ O_3, & \text{otherwise,} \end{cases} \\ C^\beta_{m,\mu} &= \begin{cases} I_3, & \text{if } \beta(\mu) = m, \\ O_3, & \text{otherwise,} \end{cases} \end{aligned} \tag{4.3}$$

for $\mu = 1, \ldots, P$ and $m = 1, \ldots, M$, where I_3 and O_3 are the 3×3 identity and null matrices. In combination with R and T, these selection matrices have the following properties:

$$\begin{aligned} RC^\alpha &= \begin{pmatrix} R^{\alpha(1)} & \ldots & R^{\alpha(P)} \end{pmatrix} \\ RC^\beta &= \begin{pmatrix} R^{\beta(1)} & \ldots & R^{\beta(P)} \end{pmatrix} \\ T^T C^\alpha &= \begin{pmatrix} T^{\alpha(1)^T} & \ldots & T^{\alpha(P)^T} \end{pmatrix} \\ T^T C^\beta &= \begin{pmatrix} T^{\beta(1)^T} & \ldots & T^{\beta(P)^T} \end{pmatrix} \end{aligned}$$

Thus C^α and C^β encode, in matrix form, the mapping between a patch m and the correspondence sets μ in which it participates.

This matrix form then allows to iteratively solve for the best rigid transformations, and only few iterations suffice to get close to the optimal solution. In our setting, we

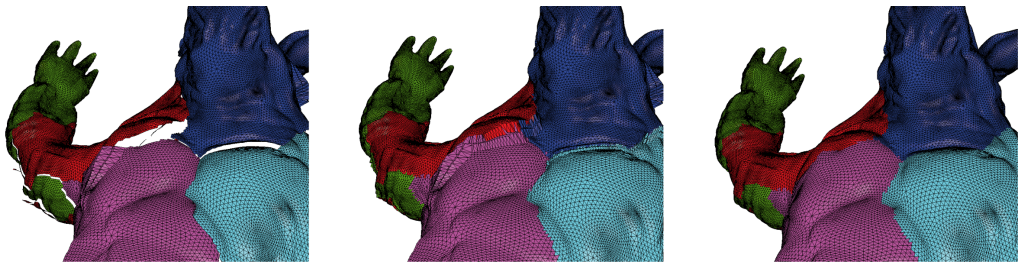

Figure 4.18. Left: The final patches before they are recombined to form the whole mesh. Middle: Simply averaging the vertex coordinates of the patch borders is not sufficient. Right: The resulting mesh, if we blend the patches as described in this section.

can actually stop the iterations rather early (after 4 passes), for we post-process the alignment anyway in order to smoothen the result (see Section 4.2.2).

Due to the iterative nature of the method, one would assume that a good starting solution is essential. Yet, we learned from our experiments that the approach is robust enough to allow for an initialization of all rotations with the identity matrix. Moreover, the result is (in principle) unique only up to a global rotation, but this can easily be adjusted by constraining one of the rotation matrices to be the identity matrix, thereby fixing the global orientation of the corresponding patch. Finally, it is possible (in case of coplanar or collinear data), that some of the resulting rotation matrices have a negative determinant and thus contain an unwanted reflection. We fix this as described in [AHB87], that is, we simply negate the last column of these matrices and continue iterating.

4.2.2 Blending Patches

Once we have computed the best rigid transformations for a set of m small patches $P^{[1]}, \ldots, P^{[m]}$ at some level of our hierarchy, it remains to blend them into a consistent larger patch P at the next coarser hierarchy level. In general, even an optimal alignment still leaves a small gap between corresponding vertices so that the patches do not fit together seamlessly in their overlap region. At first, we tried to simply average the coordinates of corresponding vertices, but this turned out to be insufficient as the alignment errors still tend to accumulate, yielding an unsatisfactory overall result for the whole mesh (see Fig. 4.18). The left picture shows the situation just before the coordinates of the final patches are averaged to form the complete mesh at the top of the hierarchy again. The picture in the middle emphasizes that simple coordinate averaging is not sufficient in this case. Instead, we decided to distribute the remaining alignment errors in a more global way as follows.

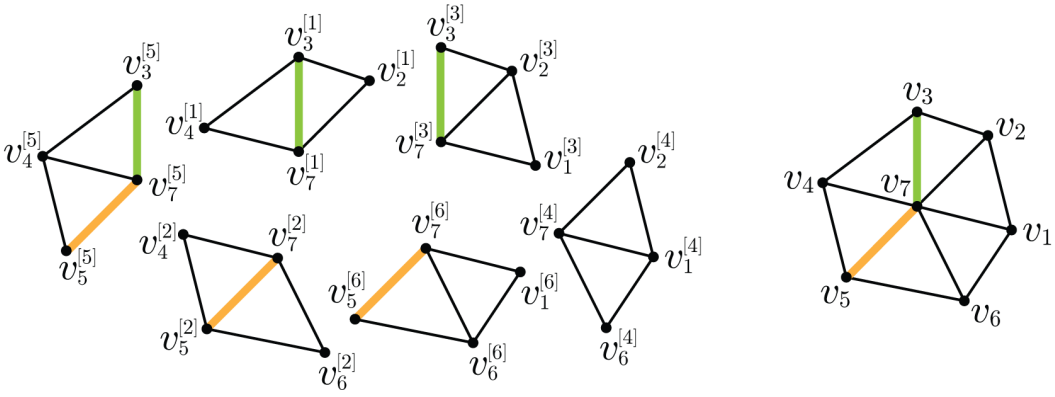

Figure 4.19. Notation for blending $m = 6$ wedges $P^{[1]}, \ldots, P^{[6]}$ (left) into a one-ring P (right) with $n = 7$ vertices. The color coded edges in the one-ring appear several times on the lower hierarchy level, which allows us to setup the linear system from Eq. (4.5).

Inspired by the handling of the consistency requirements in the deformation gradient setting [SP04], we determine the coordinates of the vertices in the large patch P such that its edges deviate as little as possible from all the corresponding edges in the small patches $P^{[k]}$. Suppose that $\boldsymbol{v} = (v_1, \ldots, v_n)$ are the vertices of P and that $[v_i, v_j]$ is one of the edges in P. Then this edge also occurs in some of the small patches $P^{[k]}$, but there it is spanned by the vertices with local coordinates $v_i^{[k]}$ and $v_j^{[k]}$ (see Figure 4.19). Ideally, the vertices $\boldsymbol{v}$ of P should be such that

$$v_i - v_j = v_i^{[k]} - v_j^{[k]} \tag{4.4}$$

for all edges $[v_i, v_j]$ in P and all its occurrences in the small patches $P^{[k]}$. Gathering all these conditions yields a linear system

$$M\boldsymbol{v} = \boldsymbol{e}, \tag{4.5}$$

where $\boldsymbol{v}$ are the unknown vertex positions of P, M is a large sparse matrix with exactly two entries 1 and -1 per row, reflecting the left hand side of (4.4), and $\boldsymbol{e}$ is the vector of corresponding edges from the small patches, reflecting the right hand side of (4.4). In general, this is an overdetermined linear system and we compute its least squares solution by solving the linear problem

$$\min_{\boldsymbol{v}} \|M\boldsymbol{v} - \boldsymbol{e}\|_2^2 \qquad \Longleftrightarrow \qquad M^T M\boldsymbol{v} = M^T\boldsymbol{e}.$$

Similar to the linear systems that appear in the work of Sumner et al. [SZGP05] and Kircher and Garland [KG08], the solution is unique only up to translation. In our setting, we resolve this by simply adding the constraint $v_1 = 0$ to the linear system.

Note that the system matrix $M^T M$ does not depend on the blending parameter t. Hence we can factorize it in a pre-processing step when building the hierarchy, so as to allow for a more efficient construction of the interpolated meshes when the user explores the shape space.

We can improve the quality of the result by replacing condition (4.4) with

$$v_i - v_j = \frac{v_i^{[k]} - v_j^{[k]}}{\|v_i^{[k]} - v_j^{[k]}\|} s_{ij}(t), \tag{4.6}$$

where $s_{ij}(t)$ is the linear interpolation of the lengths of edge $[v_i, v_j]$ in the source and the target mesh. In this way, we only keep the *orientation* of the edge from the small patch, but enforce its *desired length* on each hierarchy level.

Overall, this drastically reduces the deviation of the edge lengths in the interpolated mesh from the ideal, linearly interpolated lengths that we use at the bottom of the hierarchy to assemble the interpolated single triangles. It is remarkable to note that solving the linear system (4.5) with the conditions from (4.6) seems to yield a locally optimal solution. That is, solving the system iteratively, with the edges $v_i - v_j$ from the current solution instead of $v_i^{[k]} - v_j^{[k]}$, does not improve the edge lengths.

Moreover, this approach allows to apply additional local deformations to parts of the mesh by modifying the destination lengths. For example, we can scale parts of the mesh by tagging a set of edges and multiplying the corresponding lengths $s_{ij}(t)$ with some common scaling factor, as shown in Figure 4.28.

4.2.3 Building the Hierarchy

One question that we have not yet answered is how to get the adjacency information about the patches and how to set up the hierarchy. A common practice is to recursively decimate or cluster the mesh to construct a progressive hierarchy. Then a multi-grid method is used to propagate the solution from the coarsest up to the finest level. For example, Botsch et al. [BPGK06] solve their hierarchical shape matching problem this way. Our approach is different in that we recursively split the mesh into smaller and smaller patches, resulting in a hierarchy tree.

We start by taking the complete mesh as root node of the tree (level 0) and then descend one level by taking as many random seed triangles as we want the hierarchy to have patches on each level, say m. Using these seed triangles, we concurrently apply a region growing step until the patches meet and overlap by a triangle strip of width one. Our experiments show that it does not matter how the seed triangles are located, but theoretically it is better to place them such that the resulting patches have an irregular boundary and hence a larger overlap region. This creates the first m patches of the hierarchy tree at level one.

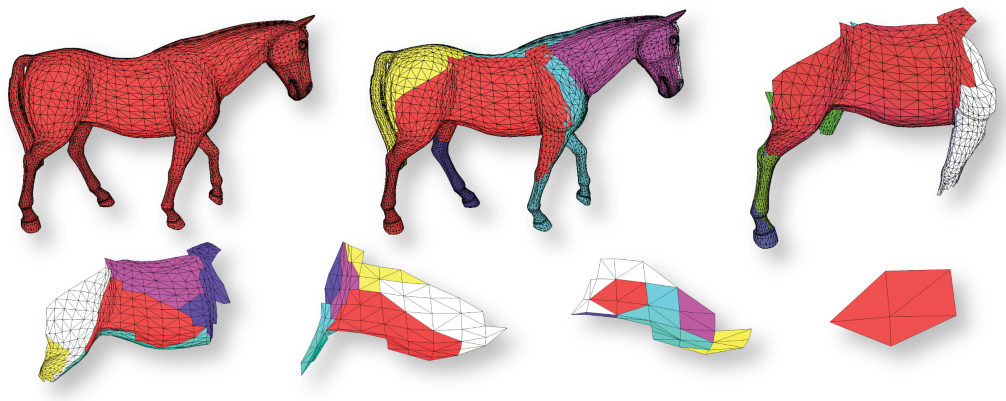

Figure 4.20. Building the hierarchy: the whole mesh (root node at level 0) is split into m patches at level 1, which in turn get split into m patches each at level 2 and so on, until the patches consist of less than m triangles (lowest level). From top left to bottom right: one branch of the hierarchy tree, where it is always the red patch whose split is shown in the next picture.

By recursively applying this scheme, we build the whole tree until we arrive at the lowest level (see Figure 4.20). Theoretically, this should be the level of triangles, but it turned out to be sufficient to stop as soon as the patches consist of less than m triangles. At this lowest level, we interpolate each of the triangles and the dihedral angles between neighbouring ones as described above and glue them together in a greedy way.

That is, we start with any of them and keep attaching the others one by one, respecting the desired dihedral angles. If such a lowest level patch is a triangle strip, then this is actually the best one can do. But if it is a triangle fan (or contains one), then this simple strategy may create gaps because the fan does not necessarily close up perfectly. However, we found that the blending procedure (see Section 4.2.2) takes care of this and smoothes these imperfections. Overall, this speeds up the interpolation process, because it reduces the number of hierarchy levels.

Although the multi-registration step (see Section 4.2.1) allows any number of patches to be aligned, we found that using $m = 6$ patches per node gives the best trade-off in terms of computation time. A smaller m creates too many hierarchy levels, and a larger m slows down the multi-registration steps, because each of them requires to compute the singular value decomposition of a $3m \times 3m$ matrix, which is created as the product of the block selection matrices C^α and C^β in (4.3) and their transposed variants. At this point the interested reader is referred to the original paper [WB00] for more information about the multi-registration method.

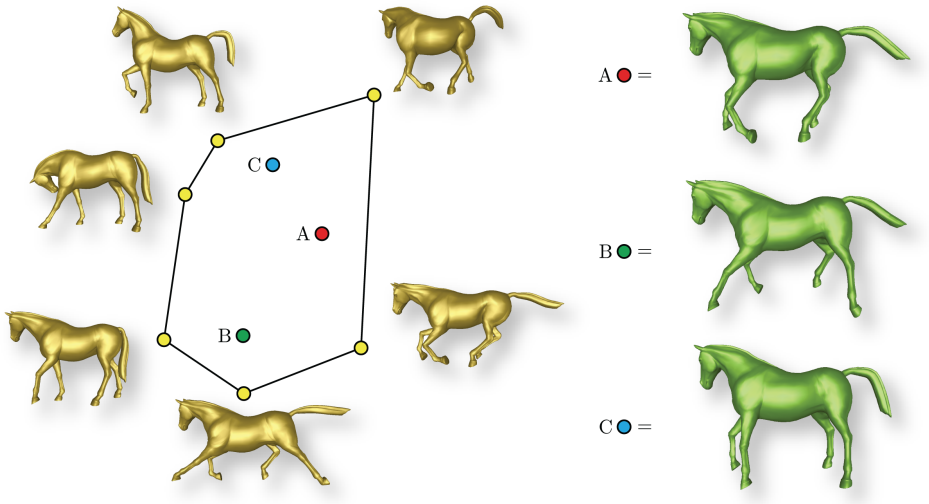

Figure 4.21. Interpolation between multiple input meshes. The n reference meshes correspond to the vertices of a control polygon in 2D, and any point within this polygon corresponds to an interpolated pose. The user can change the interpolated mesh by either moving the reference point inside the polygon, or by changing the shape of the control polygon.

4.2.4 Multiple Input Meshes

Since our approach is based on linear interpolation, it trivially allows for the interpolation between more than two input meshes. Let $a_1, \ldots, a_n$ be the lengths of a corresponding edge in n input meshes, and let $\boldsymbol{t} = (t_1, \ldots, t_n) \in [0,1]^n$ be an n-dimensional interpolation parameter. Then the interpolated edge length is $a_{\boldsymbol{t}} = t_1 a_1 + \ldots + t_n a_n$, and likewise for the interpolated dihedral angles $\alpha_{\boldsymbol{t}} = t_1 \alpha_1 + \ldots + t_n \alpha_n$. Once these values have been used to construct the two lowest levels of the hierarchy (single triangles and wedges), the remaining levels are constructed in the same way as described above. In the example shown in Figure 4.21, we use mean value coordinates [Flo03] with respect to the corners of the control polygon as interpolation parameter $\boldsymbol{t}$.

4.3 Discussion

In Section 4.2 we presented a novel method for interpolating between two or more compatible meshes which is not based on local affine transformations. Instead we linearly interpolate the intrinsic local properties of the input meshes, which is the most natural and simplest thing to do on the level of wedges. The simplicity of this idea is counterweighed by the fact that putting the wedges together such that they yield a

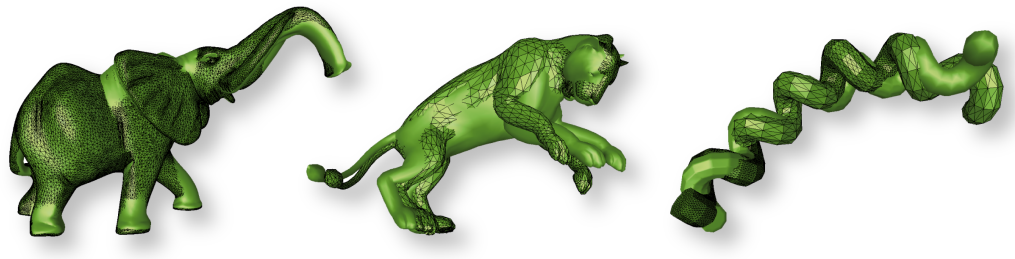

Figure 4.22. Comparing the results for interpolation parameter $t = 0.5$ between our approach (flatwire) and the results generated by the method of by Baran et al. [BVGP09]. Both methods produce a visually plausible interpolation result.

globally consistent mesh is a rather complex optimization problem. However, we found that this problem can be solved in principle by multi-registration methods and since this can be combined with a hierarchical decomposition of the mesh in larger and larger patches, it can actually be solved efficiently as well. The presented method is able to produce results comparable in quality to recent state-of-the-art approaches [KMP07; KG08; CL09], but our method is significantly simpler and faster. The only method that can compete in terms of speed, is the one by Kircher and Garland [KG08], but as mentioned above we can handle much larger meshes, because our mesh hierarchy decomposes them into digestible chunks.

Furthermore, Figure 4.22 underlines one major problem in this setting. Due to the lack of a widely accepted benchmark which is able to measure the *quality* of mesh interpolation methods, the results are always subject to the user's interpretation and requirements of a "plausible" motion between the source and the target configuration of the mesh. Creating such a benchmark would be utterly hard, if not even impossible. What would be the ground truth for an interpolation where we only have access to the mesh vertex data and the connectivity? To be able to bridge the gap between *visually plausible* and *physically correct*, we require more properties of the mesh. But depending on the object that is reflected by the mesh, these properties could be hard to capture or they might not even exist.

Figure 4.23 confirms that our approach generates basically the same interpolation results as the ones by Kilian et al. [KMP07] and Chu et al. [CL09], but we are at least an order of magnitude faster (see Table 4.1). The armadillo example emphasizes that very large meshes do not pose a challenge for our method, because of our hierarchical structure. The cylinder deforming into a helix probably illustrates best what we are striving for with this method. The interpolation follows the multiple rotations (far greater than 180 degrees) in a visually plausible way, due to the fact that the multi-registration step takes care of the global rotation. Note that the result shows a remarkable resemblance

Figure 4.23. Interpolation between two input meshes (leftmost and rightmost column). The interpolated poses (in green) are shown for parameter values of $t = 0.25$, $t = 0.5$, $t = 0.75$.

to the outcome of the physical simulations of discrete elastic rods [BWR+08] and does not suffer from the fact that the triangle density varies strongly over the mesh.

The left picture in Figure 4.24 illustrates how little the edge lengths in the interpolated mesh deviate from the desired lengths that we impose at the bottom of our hierarchy. The plots show the maximum and the minimum relative difference from the linearly interpolated lengths. While the global worst case is about -12% the major fraction of the edges (99%) do not differ by more than 2.5% from the ideal value. This confirms that our alignment and blending steps affect the local metric only very

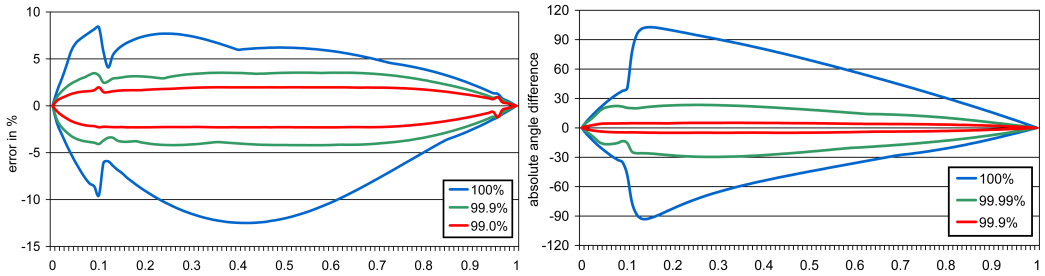

Figure 4.24. *Left*: relative error between the effective edge lengths in the interpolated mesh and the ideal linearly interpolated lengths for the elephant example in Figure 4.27. Plotting this error for all edges yields the envelope represented by the blue curves. Neglecting the 0.1% edges with the worst deviation results in the green envelope, and the red one visualizes the envelope of 99% of all edges. *Right*: relative error between the effective dihedral angles and the linearly interpolated ones for the elephant example. Compare to Figure 4.24.

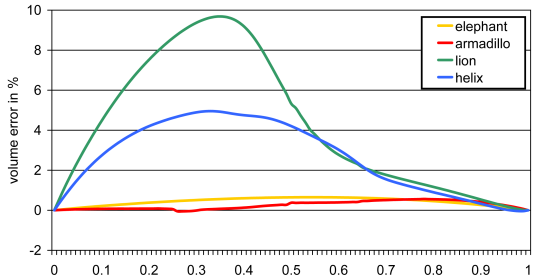

Figure 4.25. Relative error between the volume of the interpolated mesh and the volume of the input meshes for the examples in Figures 4.23 and 4.27.

slightly. The picture on the right of Figure 4.24 shows the equivalent plots for the angles, but this time on an absolute scale. Although the overall worst case differs by more than 100 degrees from the ideal value, such extreme deviations are very rare to happen. For 99.9% of all edges, the interpolated dihedral angle lies within ± 5 degrees from the linearly interpolated angle, and the maximum error of 99% of the angles is less than 1 degree. Again this shows how well our global alignment procedure keeps the dihedral angles that we impose on the wedge level.

Interestingly, our method also does a good job in preserving the volume of the meshes during interpolation as shown in Figure 4.25, although we do not directly consider this as a constraint during our reconstruction.

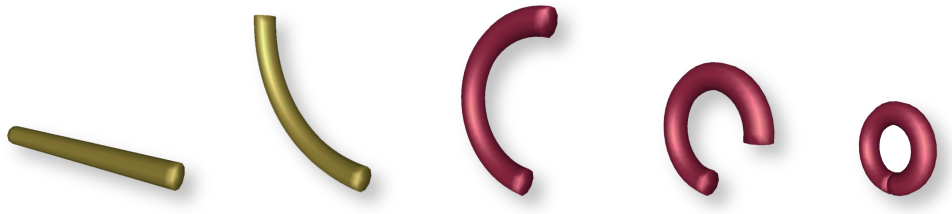

Figure 4.26. Extrapolation of a cylinder. The two input meshes and extrapolated poses for parameter values of $t = 2$, $t = 3$, $t = 4$ (from left to right).

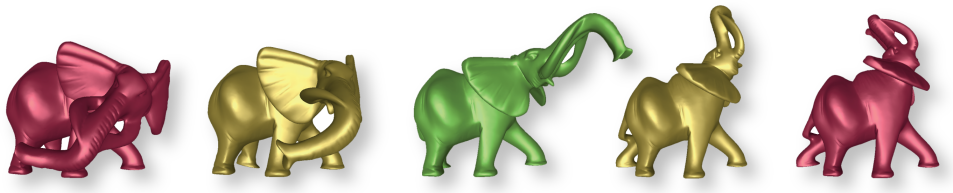

Figure 4.27. Extrapolating the elephant. The parameter values of the meshes are $t = -0.25$, $t = 0$, $t = 0.5$, $t = 1$, $t = 1.25$ (from left to right).

4.3.1 Extrapolation and Constrained Interpolation

In principle, our method can also be used for extrapolating between two or more input meshes, that is, the interpolation parameter can be chosen outside the range $[0, 1]$. But then it is no longer guaranteed that the interpolated edge lengths match up to form a triangle (an interpolated edge can end up being negative or bigger than the sum of the other two), and the interpolated dihedral angle may leave the valid range between -180 and $+180$ degrees. However, we found that this happens only for rather extreme extrapolations and works well in most cases (see Figure 4.26 and 4.27). For the elephant example in Figure 4.27 we cannot go much beyond the interval $[-0.25, 1.25]$, but if the edge lengths and dihedral angles in the input meshes are quite similar, then we can actually extrapolate quite far as shown in Figure 4.26. Figure 4.28 shows an example of a constrained interpolation as explained at the end of Section 4.2.2. We applied an additional scaling factor of $t + 1$ to the lions head, tail and paws (i.e., the scaling factor varies linearly from 1 at the source mesh to 2 at the target mesh).

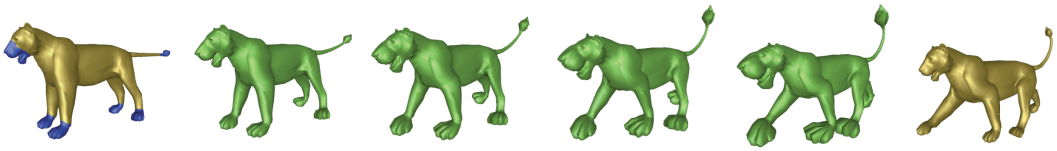

Figure 4.28. Example of a constrained interpolation for the lion using a scaling factor of $2t$ for the edges in the selected regions (blue). The parameter values of the interpolated meshes are $t = 0.25$, $t = 0.5$, $t = 0.75$, $t = 1$ (from left to right).

meshes			pre-processing		mesh interpolation		
	vertices	faces	split	factor	interpol.	reg.	blend.
cylinder	312	620	2.94	59.71	1.17	18.78	5.21
helix	1212	2420	12.75	243.76	3.98	84.11	24.66
lion	5000	9996	127.55	1248.22	16.05	365.99	122.45
horse	8431	16843	300.41	2140.96	72.45	544.01	205.93
elephant	39969	79946	5975.31	11954.33	126.15	2632.40	1327.28
armadillo	165954	331904	90106.71	54339.12	570.07	11273.22	7105.04

Table 4.1. Timings for all meshes shown in this chapter, measured in milliseconds

4.3.2 Timings

The timings in Table 4.1 report that the preprocessing step is the computationally most intense part of the approach, but still reasonable even for large meshes. Once this work is done, constructing an interpolated pose is not too expensive. The cost for interpolating the edge lengths and dihedral angles at the bottom of the hierarchy is negligible and the registration time grows linearly with the number of triangles. From a certain mesh size on, the blending step becomes the most expensive part of the pipeline, since the matrices from Equation (4.4) are then relatively large on the top of the hierarchy. All timings were measured on an Intel Core2 Duo Laptop with 4 GB RAM and a 2.5 Ghz CPU.

4.3.3 Robustness

In Figure 4.29 we applied a reasonable amount of noise (2% of the bounding box diagonal) to the elephant model from Fig. 4.27. The figure illustrates that the resulting interpolation is not significantly influenced. Of course it is possible to break the method by adding more noise, since then the corresponding patches cannot be aligned correctly any longer. The purpose of Figure 4.30 is to emphasize that the method is also robust

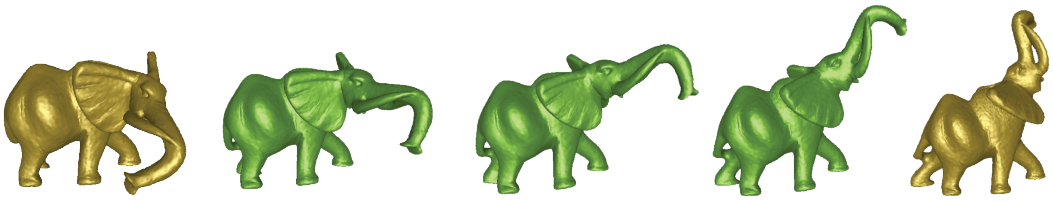

Figure 4.29. Interpolating the elephant with additional noise (compare with Figure 4.27). The parameter values of the interpolated poses are $t = 0.25$, $t = 0.5$, and $t = 0.75$ (from left to right).

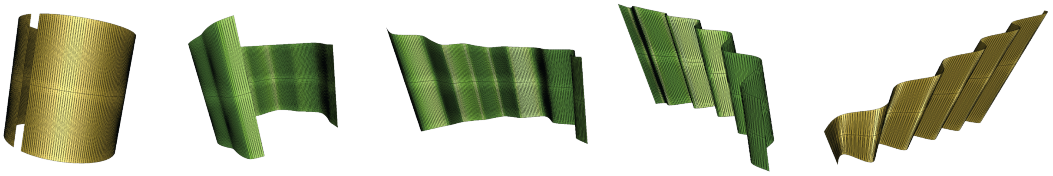

Figure 4.30. Interpolating long and skinny triangles. The parameter values of the interpolated poses are $t = 0.25$, $t = 0.5$, and $t = 0.75$ (from left to right).

t	$m = 3$	$m = 6$	$m = 9$
0	$2.202 \cdot 10^{-3}$	$2.201 \cdot 10^{-3}$	$2.191 \cdot 10^{-3}$
0.25	$5.206 \cdot 10^{-3}$	$6.101 \cdot 10^{-3}$	$5.933 \cdot 10^{-3}$
0.5	$6.549 \cdot 10^{-3}$	$8.876 \cdot 10^{-3}$	$9.295 \cdot 10^{-3}$
0.75	$5.781 \cdot 10^{-3}$	$8.537 \cdot 10^{-3}$	$8.544 \cdot 10^{-3}$
1	$1.938 \cdot 10^{-3}$	$1.925 \cdot 10^{-3}$	$2.056 \cdot 10^{-3}$

Table 4.2. Hausdorff distance between N_t and M_t (see Fig. 4.31) for different parameters of t and m.

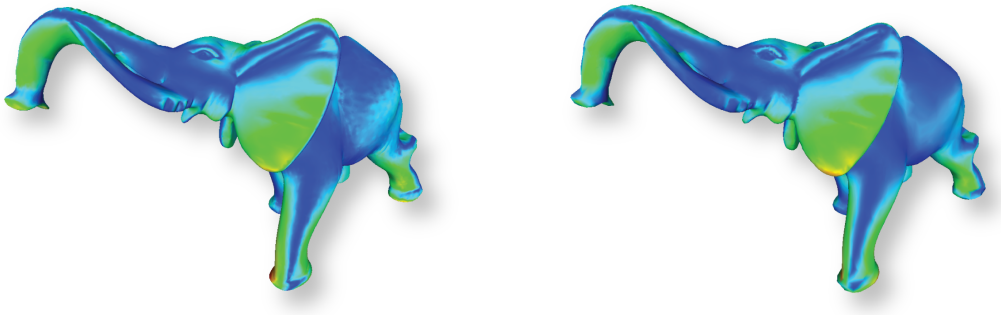

Figure 4.31. The color coded Hausdorff distance between N_t and M_t for $t = 0.5$ measured with the Metro tool [**?**]. The input meshes for the interpolation are shown in Figure 3.15, p. 41.

enough to handle long and skinny triangles. For further investigation on the influence of a model's triangulation we used our mesh massage framework to generate two different sets of input meshes for the interpolation. The first set of meshes N_1 and N_2 contains the non-modified version of the elephant, while the meshes M_1 and M_2 in the second set share a significantly different triangulation of the elephant's geometry (see Sec. 3.3.4 p. 39). In the next step we measured the Hausdorff distance between the resulting interpolated meshes N_t and M_t. Table 4.2 summarizes the results for different interpolation parameters t as well as for different numbers of patches m, used in the splitting/combining step. We see that the distance between the meshes at the beginning of the interpolation is already $\approx 2.2 \cdot 10^{-3}$ (normalized to the bounding box) for all values of m. This is due to the fact, that we applied several subdivision steps and quadric edge collapses (see Sec. 3.3.4). This also explains the distance at the end of the interpolation for $t = 1$.

The maximum Hausdorff distance is always reached for $t = 0.5$ with $6.549 \cdot 10^{-3}$ (respectively $8.876 \cdot 10^{-3}$ and $9.295 \cdot 10^{-3}$), because here we are combining edges and angles from both input meshes with equivalent weighting. The blending step (see Sec. 4.2.2) has the highest influence at this point. Figure 4.31 shows the color coded Hausdorff distance between the resulting interpolated meshes N_t and M_t for $t = 0.5$ and $m = 6$.

Finally, Figure 4.32 shows an example of a rather tricky setting for our interpolation method. The movement of the plane from M_0 to M_1 is not that difficult, since the plane basically scales and bends. For example, deformation gradients perform well in this setting, since there are no large rotations involved. Why does this example pose a challenge to our method then? The difficulty arises from the highly irregular tessellation of the meshes which is depicted in the magnified window in the upper

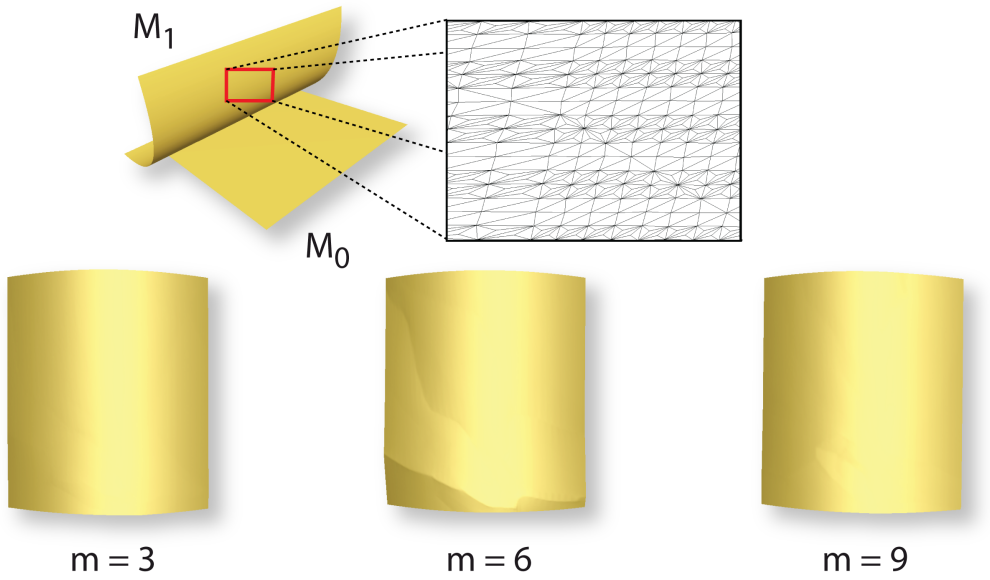

Figure 4.32. Interpolation of highly irregular tessellated meshes. The lower row reflects the results for different patch sizes used in the hierarchy. The interpolation parameter is $t = 0.5$ for all meshes.

part of the Figure. If we compare the result for $m = 3$ patches of our method to the outcome of the deformation gradient interpolation, we see that the Hausdorff distance ($3.115 \cdot 10^{-3}$) between both results is negligibly small. But for $m = 6$ patches the distance is already increased by an order of magnitude ($1.110 \cdot 10^{-2}$) and our method produces visible artifacts. The creases that become visible reflect the borders of the final patches. While for all other results in this chapter changing our standard setting of $m = 6$ remaind without noticeable effect, the effect becomes quite dramatic in this specific situation due to the following reasons: Firstly, as mentioned in Section 4.2.1 and Section 4.2.2 the patches in the hierarchy are constructed such that they overlap by a triangle strip. This overlap has the width of one triangle and in this specific case, it seems not to be sufficient to produce the expected result, due to the irregular structure of the mesh. If we increase the number of patches to $m = 9$ the creases become far less pronounced, because the partition of the mesh into said patches produces different borders and they seem to add up better in this case. Secondly, changing the number of patches also changes the form of our hierarchy tree. While we have a rather wide and flat tree for $m = 9$, we create more hierarchy levels for $m = 3$ in a taller tree and because a smaller number of patches corresponds to more gluing steps (see Eq. (4.4)) the highly irregular tessellation of this mesh, the result benefits from the higher number of averaging steps.

Chapter 5

Compact Representations

In the previous chapter we focused on the task of creating motion out of a few given meshes. This chapter now reverses that idea. Assume that we already have a given animation sequence. As previously mentioned in Section 2.4, many sophisticated methods exist for compressing either static or dynamic meshes (i.e. animated meshes), yet animated meshes could obviously benefit from a *thinning* step, prior to the actual compression step (see Fig. 5.1).

If this thinning identifies the most representative meshes for the given sequence, then by computing linear combinations of those carefully selected meshes, the original sequence can be represented with minimal approximation error. In other words, we would like to find a method that provides us with the possibility to identify patterns in the data and furthermore emphasizes the similarities and differences inherent in the data. Probably the most famous method for this task was introduced by Pearson [Pea01] in 1901. It is commonly known as Principal Component Analysis (PCA). Therefore, we will briefly review it and outline how to apply the PCA in terms of thinning an animation sequence.

Figure 5.1. A compact representation of an animation sequence can be created by *thinning* the sequence to the most important meshes. Reconstruction of the original sequence is carried out by computing linear combinations of those meshes.

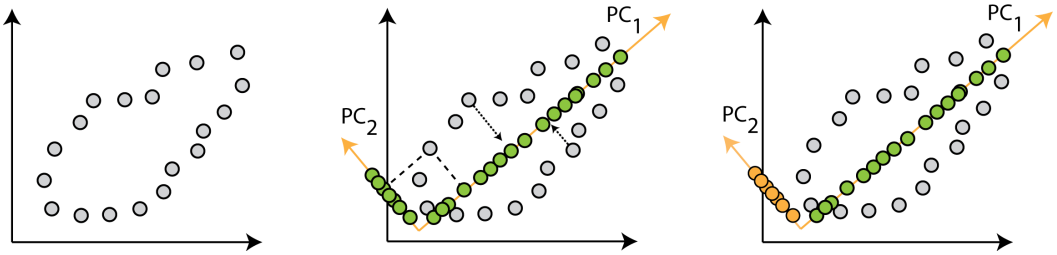

Figure 5.2. Given a point cloud in $\mathbb{R}^2$ (left) the PCA sets up a new orthonormal system, such that the first Principal Component (PC_1) captures the direction of the highest variance in the point cloud and respectively PC_2 represents the second biggest variance. Using linear combinations of PC_1 and PC_2 the data can be reconstructed without error. If we drop the Principal Components (right) which have lesser significance (in this case PC_2) the data still can be reproduced, but with error. Yet, the error is minimal in the 2-norm.

5.1 Principal Component Analysis

Consider the example in Figure 5.2, given a set of points $\boldsymbol{p} = (p_1, \ldots, p_n)$ with $p_i \in \mathbb{R}^2$, the PCA identifies the directions of high variance in the data set. Briefly speaking, the PCA is an orthogonal transformation of the data to a new coordinate system (see Fig. 5.2, middle), such that the direction of the greatest variance in the data set is reflected by the first axis and the second biggest by the second axis (the extension of this concept to higher dimensions is straightforward).

In our case we exploit the fact that the axes of the new coordinate system, commonly referred to as Principal Components (PCs), encode the direction with significant variance in the data set. We can represent the data lossless, i.e. without any error, by using as many PCs as we have dimensions, because this corresponds to a plain rotation of the original coordinate system. Hence, the critical observation is the *decreasing* significance of the PCs, if their number *increases*. Therefore, if we drop the less significant PCs, we can still express $\boldsymbol{p}$ and introduce just a small error in the reconstruction process. This is sketched in the right picture of Figure 5.2, where we neglect PC_2 and represent each point from the set $\boldsymbol{p}$ only by PC_1 in combination with an appropriate scalar value, gained through projection of the original data point onto PC_1. Thus, we found a representation of $\boldsymbol{p}$ in a lower dimensional space and because the PCs encode the variance in the set, we have only introduced minimal error in terms of the 2-norm. How can we apply this knowledge to mesh animations now?

Suppose we have a sequence of meshes $\boldsymbol{m} = (m_1, \ldots, m_n)$ with $m_i \in \mathbb{R}^d$. Then we can rearrange the x, y, z coordinates of the v vertices for each mesh slightly and combine them in a single vector, such that $d = 3v$ in this case

$$m_i^T = \left(x_{i,1} y_{i,1} z_{i,1}, \cdots, x_{i,v} y_{i,v} z_{i,v}\right) \in \mathbb{R}^{3v}. \tag{5.1}$$

Now, if we change our perspective slightly and do not understand the m_i as 3D meshes, but interpret them as *points* in a d-dimensional space, this is a very similar setting compared to the previous example in Figure 5.2. If we further assume that all m_i have the same connectivity and only the vertex positions change over time, this allows us to represent the whole animation sequence in matrix form as $M = (m_1 \cdots m_n) \in \mathbb{R}^{d \times n}$, e.g. shown by Alexa and Müller[AM00] or Karni and Gotsman [KG04]. Each column of M represents the geometry information of one mesh m_i and the number of rows in M corresponds to three times the number of vertices. If we proceed straight by the book, the next step would be to compute the covariance matrix $C \in \mathbb{R}^{d \times d}$

$$C = \frac{1}{n} \sum_{i=1}^{n} (m_i - \bar{m})(m_i - \bar{m})^T \tag{5.2}$$

where

$$\bar{m} = \frac{1}{n} \sum_{i=1}^{n} m_i \tag{5.3}$$

denotes the mean of all meshes in the sequence. Subtracting it from each mesh allows us to write Eq. (5.2) as

$$C = \frac{1}{n} M M^T. \tag{5.4}$$

We are interested in the covariance matrix, because the covariance between each pair of meshes is a measure of the linear coupling between them. Therefore, the eigenvectors, corresponding to the largest eigenvalues of C, provide us with the directions of the significant variation in the data set. To compute the eigenvectors and eigenvalues, we apply the Singular Value Decomposition (SVD) to M and decompose it into $M = U\tilde{S}V^T$ where $U \in \mathbb{R}^{d \times d}$ and $V \in \mathbb{R}^{n \times n}$ are orthonormal matrices and $\tilde{S} \in \mathbb{R}^{d \times n}$ is of the form

$$\tilde{S} = \begin{pmatrix} S \\ 0 \end{pmatrix} \tag{5.5}$$

where $S \in \mathbb{R}^{n \times n}$ is a diagonal matrix which contains the singular values of M. This allows us to rewrite Eq. (5.4) as

$$C = \frac{1}{n} M M^T = \frac{1}{n} U\tilde{S}V(U\tilde{S}V)^T = \frac{1}{n} U\tilde{S}VV^T\tilde{S}^T U^T = \frac{1}{n} US^2U^T$$

but this is the eigendecomposition of the symmetric matrix C and thus we recognize the columns of U as the eigenvectors corresponding to the squared singular values of M, i.e. the eigenvalues (see e.g. [SAG+05]). If we denote $\tilde{U} \in \mathbb{R}^{d \times k}$ as the matrix of the first k eigenvectors, sorted according to the descending order of their corresponding eigenvalues, then we can express the projection of the sequence M onto the first k eigenvectors (PCs) as $P = \tilde{U}^T M$ with $P \in \mathbb{R}^{k \times n}$. In other words, we are now able to express the sequence in a compact form through the k principal components

Figure 5.3. The first three Principal Components extracted from the animation sequence, shown in Figure 5.1.

in combination with the $k \cdot n$ weights stored in P instead of using the $d \cdot n$ values of M. The principal components are merely the basis vectors for a k-dimensional subspace of our initial $\mathbb{R}^d$. Please note that throughout the rest of this chapter we do not use the standard PCA as described above, but a slightly modified variant proposed by Roweis [Row98]. The main difference is the utilization of an expectation maximization algorithm that allows a very efficient extraction of a few eigenvectors and eigenvalues from a large data set. The key idea of this algorithm is to execute the following computations iteratively

$$\begin{aligned} X &= (P^T P)^{-1} P^T M \\ P^{\text{new}} &= M X^T (X X^T)^{-1} \end{aligned}$$

where M is our matrix storing the meshes and $X \in \mathbb{R}^{k \times n}$ is an auxiliary matrix. After sufficiently many iterations (we use 20 steps as suggested by Roweis [Row98]) the columns of $P \in \mathbb{R}^{d \times k}$ store the first k principal components. Please note that P can be initialized randomly at the beginning of the algorithm (see [Row98] for further details).

5.1.1 Principal Component Representation

At the beginning of this chapter we introduced the idea of a thinning method. Our requirements were twofold, firstly the method should be able to identify the most representative meshes for a given animation sequence and secondly the original sequence should be representable by linear combinations of those representative meshes with the smallest possible error. In theory and as the previous section suggests, at least for the second requirement the PCA provides a valid solution. But does the PCA or more precisely, do the Principal Components fulfill the first requirement equally well? Figure 5.3 shows the first three PCs extracted from the sequence in Figure 5.1. It is rather obvious, that although the PCs allow the reconstruction of the whole sequence $\boldsymbol{m}$ with minimal error, the PCs themselves are not representable for the sequence at

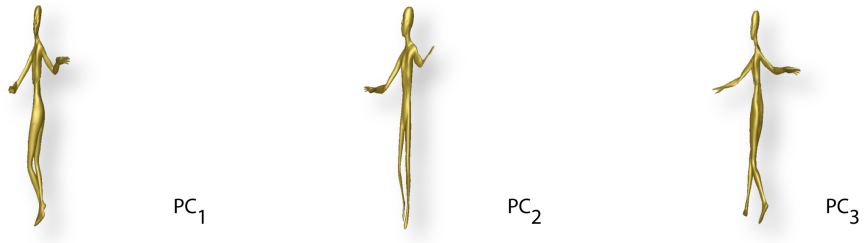

Figure 5.4. The same Principal Components as shown in Figure 5.3, but this time we added the average mesh of the sequence to each PC.

all. Let us briefly recollect the steps necessary to represent $\boldsymbol{m}$ with a reduced set of PCs and unveil what the reason is:

- set up the matrix M from the sequence $\boldsymbol{m}$
- subtract the sequence mean (Eq. (5.3)) from each column of M
- compute the Principal Components
- project each column of M onto the PCs, i.e. compute $P = \tilde{U}^T M$
- each column in M can be reconstructed as a linear combination of $\tilde{U}$ and the corresponding entries in P
- adding back the mean (Eq. (5.3)) to each column finally yields the reconstructed mesh

We realize from the last step that in the same way as we have to add the mean back to each column to create the final mesh, we also have to add it to the PCs if we would like to see which share of the sequence the Principal Components actually encode. As Figure 5.4 confirms, this causes the PCs at least to be identifiable as meshes and also makes them more suitable as input to the methods mentioned in Section 2.4.

Yet, since the original sequence incorporates a significant amount of rotation, the average (mean) mesh itself has a starving appearance to it, which is unfortunately propagated to the Principal Component meshes as well. This is clearly not the kind of thinning we initially had in mind.

The visual coupling between the extracted PCs and the original meshes in the sequence is rather loose, because the PCA requires only the linear combinations of the PCs to be meaningful, i.e. to reproduce the original meshes as good as possible. There is no need for the PCs to be meaningful themselves. Therefore, the next section introduces a different approach to find the most prominent meshes from a given animation sequence.

5.2 Neural Gas

In this section we present a fast and simple framework that approaches the problem of reducing the amount of data in mesh animations from a slightly different angle. Our framework automatically clusters the meshes into groups representing similar deformations. We show that it is possible to reconstruct the whole sequence from these key-frames with negligible error (Section 5.2.3). Since each center of a cluster is a member of the underlying manifold of meshes, the set of *key-frames* extracted by our framework fits excellenty as input to *any* state-of-the-art compression algorithm mentioned in Section 2.4.

We consider a key-frame k_j to be *meaningful* if it lies on the same data manifold as the input meshes m_i or at least close to it, in contrast to the PCA eigenvectors which do not satisfy this property; see Figure 5.3 and 5.4. Although this restriction gives us less degrees of freedom as compared to PCA, we still need only slightly more key-frames to obtain comparable reconstruction results (see Section 5.2.2).

Key-frames are further considered *optimal* if they allow to reconstruct all meshes of the sequence with minimal global approximation error, measured in the 2-norm. Optimal key-frames can thus be found by minimizing a quadratic cost function, and without any other constraints this yields exactly the PCA solution described in the previous section. But if we also want the key-frames to be meaningful, we end up with a rather complex optimization problem.

However, we can solve a simpler problem instead and still get reasonable results. Usually the data manifold is highly non-linear and the linear combination of meaningful key-frames can lie outside the manifold. As a consequence, meshes from the original sequence can only be obtained by linear combinations of close-by key-frames. Therefore, the overall approximation error is also going to be small if we minimize the standard quantization error [Har75] instead, i.e. the squared Euclidean distance of all m_i to their respective closest key-frame.

The quantization error constitutes a classical objective of clustering algorithms and one of the most popular clustering algorithms which is directly based on this cost function is the k-means algorithm [Har75]. For instance, it is used by Park and Shin [PS04] for example-based motion cloning. However, it is well known that k-means clustering is highly sensitive to initialization and usually finds only local optima of the cost function, i.e. suboptimal key-frames. Therefore, we use an efficient alternative that optimizes the same cost function as k-means clustering in the limit, but does not suffer from the problems of k-means. Our method is based on the Neural Gas (NG) algorithm by Martinetz et al. [MBS93]. NG is a vector quantization technique that aims to represent given data (i.e. meshes) $M \subseteq \mathbb{R}^d$ faithfully by prototypes (i.e. key-frames) $k_j \in \mathbb{R}^d$, $j = 1, \ldots, l$. For a continuous input distribution given by a probability density function

Figure 5.5. Three key-frames extracted from the sequence shown in Figure 5.1 by Batch-Neural-Gas. Since they lie on the data manifold, the key-frames represent the sequence much better, compared to the PCA approach (see Fig. 5.3 and Fig. 5.4).

$P(m)$ over M, the cost function minimized by NG is

$$E \sim \frac{1}{2} \sum_{j=1}^{l} \int h_\lambda\big(\mathrm{rk}(k_j, m)\big) \|m - k_j\|^2 P(m)\, dm,$$

where $\mathrm{rk}(k_j, m) = \#\{k_i : \|m-k_i\| < \|m-k_j\|\}$ denotes the rank of the key-frame k_j arranged according to the distance from the mesh m, i.e. it indicates whether key-frame k_j is representative for m (corresponding to rank 0) or not (corresponding to a large rank). The parameter $\lambda > 0$ controls the neighborhood range, i.e. the area of influence of each single point, through the exponential function $h_\lambda(t) = \exp(-t/\lambda)$. This important parameter is initialized with a high value and then quickly driven asymptotically to zero, yielding the characteristic dynamics of NG. For $\lambda \to 0$ the standard quantization error is recovered in the limit. By integrating the differences of the key-frames according to all meshes in the beginning and weighted according to the ranks, NG is not sensitive to initialization and able to overcome local optima, in contrast to the popular k-means algorithm.

Because the final key-frames k_j found by NG lie on the data manifold (it has been shown in [MS94] that the final NG solution can be extended to a valid Voronoi tessellation of the given manifold under mild conditions on the density of the mesh sequence), they are similar to meshes of the sequence (see Fig. 5.5). Due to the cost function E, a further benefit of NG can be observed: it has been shown in [MBS93] that the so-called magnification factor of NG approaches 2/3. Roughly speaking, the magnification factor characterizes the fraction of meshes from the original sequence represented by one key-frame depending on the underlying density of the sequence. Because of this factor, NG focusses on regions which are only sparsely covered, while key-frames in dense regions represent a higher percentage of meshes. Thus, NG is able to adequately capture regions of the mesh sequence with large deformations.

In the original formulation NG optimizes its cost function in an online mode by using a stochastic gradient descent method. That means, that meshes are presented

several times in random order to the algorithm and adaptation of the key-frames takes place after every single mesh. For that reason, a huge number of training steps is necessary for convergence [MBS93]. However, for a given finite sequence $\boldsymbol{m} = (m_1, m_2, \ldots, m_n)$ the cost function of NG becomes

$$E \sim \frac{1}{2} \sum_{j=1}^{l} \sum_{i=1}^{n} h_\lambda\big(\mathrm{rk}(k_j, m_i)\big) \|m_i - k_j\|^2.$$

For this special setting, Cottrell et al. [CHHV06] introduced a fast batch optimization technique that adapts key-frames according to all meshes at once, similar to the original k-means adaptation scheme. The resulting Batch NG algorithm (BNG) determines first the ranks $r_{ji} = \mathrm{rk}(k_j, m_i)$ for fixed key-frames k_j and then new key-frames via the update formula

$$k_j = \frac{\sum_{i=1}^{n} h_\lambda(r_{ji}) \cdot m_i}{\sum_{i=1}^{l} h_\lambda(r_{ji})}$$

for fixed ranks r_{ji}. BNG shows the same accuracy and behaviour as NG, but its convergence is quadratic instead of linear as for NG. This scheme subsequently determines the responsibility of key-frames for meshes of the sequence by means of the ranks. Afterwards, it calculates new key-frames as the generalized mean of the meshes, weighted according to the responsibilities.

5.2.1 Finding the Interpolation Weights

Now that we have found a set of meaningful and optimal key-frames, the next goal is to find for each mesh m_i the *weights* $\lambda = (\lambda_1, \ldots, \lambda_l)$ so that the reconstructed mesh

$$\hat{m}_i = \sum_{j=1}^{l} \lambda_{ij} k_j \tag{5.6}$$

is as close to the original m_i as possible. Denoting by $K = (k_1 \cdots k_l) \in \mathbb{R}^{d \times l}$ the matrix with key-frames k_j as columns, we have $\hat{m}_i = K\lambda$ and the best set of weights is found by

$$\min_{\lambda} \|m_i - \hat{m}_i\|,$$

which is equivalent to solving the normal equation

$$K^T K \lambda = K^T m_i.$$

5.2.2 Reconstruction Quality

Instead of the d_a-error (Eq. 5.7), which was introduced by Karni and Gotsman [KG04] and is therefore often referred to as the KG-error, we measure the quality of our reconstruction using the root mean square error [CRS98], which was also used in [BSM+03]

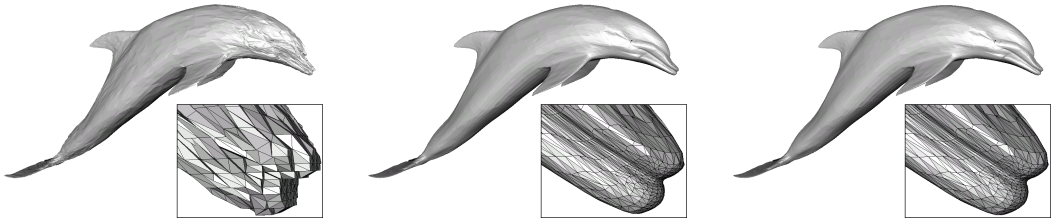

Figure 5.6. Comparing the KG-error: quantization using 8 bits (left), original model (middle), reconstruction using the key-frames extracted with our approach (right).

Model	# KF (basis vectors)	Our approach KG-error	PCA approach KG-error
Dolphin	100	0	0.024
	50	0.029	0.029
	25	0.094	0.075
	10	0.975	0.607
Face	200	0.062	0.046
	100	0.131	0.081
	50	0.254	0.146

Table 5.1. Using the key-frames extracted by BNG as reconstruction basis, we get an error similar to the one achieved by PCA [KG04].

and [GK04]. As already noted by Guskov and Khodakovsky [GK04] the KG-error does not capture the visual quality of the reconstruction well.

In Figure 5.6, the approximations of the dolphin model (left and right) have the same KG-error with respect to the original (middle), but the model on the left shows the well-known staircase artifacts [GK04] caused by quantization of the vertex positions. We only measure the KG-error in Table 5.1 to show that although our selected key-frames are not optimal in the sense of the 2-norm, they still introduce only a slightly larger error, compared to the PCA result. The KG-error in this case is defined as

$$d_a = 100 \, \frac{\|M - \hat{M}\|}{\|M - \tilde{M}\|}, \tag{5.7}$$

where the matrix $\hat{M} = (\hat{m}_1 \cdots \hat{m}_n) \in \mathbb{R}^{d \times n}$ stores the reconstructed meshes $\hat{m}_i$ in each column. The matrix $\tilde{M} = (\tilde{m}_1 \cdots \tilde{m}_n) \in \mathbb{R}^{d \times n}$ is an "average" matrix, whose columns consist again of the x,y,z coordinates, but averaged for each m_i (see [KG04] for further details).

Figure 5.7 shows the reconstruction error for the a swimming dolphin, for a walking chicken that flees in panic at the end of the sequence and for the animation of

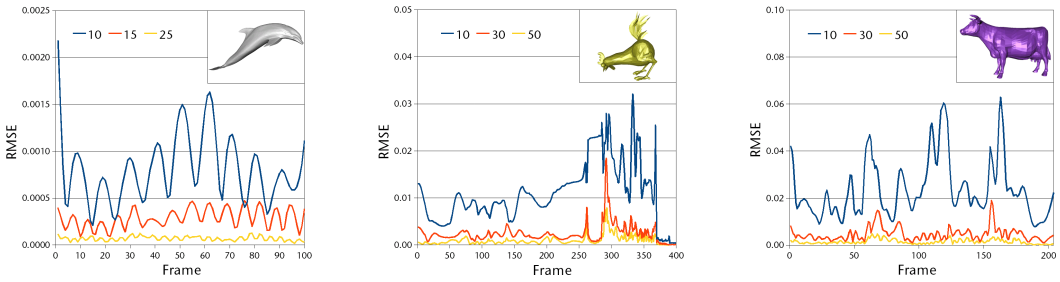

Figure 5.7. Error distribution relative to the number of key-frames, for the dolphin (left), chicken (middle), and cow sequence (right).

a cow which is tossed into the air and subject to further severe deformations. Since the motion in the dolphin sequence is sine-like, it has a repetitive character and only moves in two dimensions, which fits well into our framework. We thus get almost as good results as [KG04]; see Table 5.1.

For the sequence of the walking chicken it is obvious that 10 key-frames are insufficient to capture the highly non-linear motion. But using 30 key-frames allows us to halve the error at the end of the sequence. This is due to the fact that most of the motion is present at the end of the sequence when the chicken starts to panic and our framework can adapt perfectly to this by selecting more key-frames to represent the end.

The cow sequence shows a different behavior. Right from the start of the sequence the cow is subject to extreme transformations. Using only 10 key-frames, the three times when the cow is dragged into the air are clearly reflected by the error. The frames in between are already captured well. With 30 key-frames we can also capture these parts well, leading to a reduction of the error by at least two thirds.

Please note that Figure 5.7 should only emphasize that our framework has the potential to extract well suited key-frames from a dynamic mesh. This is indirectly visible if the number of frames for the chicken and the cow sequence are compared. For both sequences 30 key-frames are sufficient, which is remarkable since the chicken sequence has twice as many frames, but most of the motion happens at the end, leading to the conclusion that our framework extracts suitable key-frames in this case. A comparison to a state-of-the-art approach for extracting key-frames is given in Table 5.3.

5.2.3 Results

We compare our framework to the recently introduced approach of Lee et al. [LLWC08] since we consider it an extension of [HCHY05], which is basically a comparison between the most promising algorithms for extracting key-frames. We also use the peak

Figure 5.8. Selected frames from the dolphin, chicken, cow and face sequence which are used within this section.

signal-to-noise ratio

$$\mathrm{PSNR}(m_i, m_j) = 20 \log_{10} \left(\frac{\max(\mathrm{Diag}(m_i), \mathrm{Diag}(m_j))}{\mathrm{RMSE}(m_i, m_j)} \right),$$

where $\mathrm{Diag}(m_i)$ is the bounding box diagonal of the mesh m_i, which measures the quality of a signal that is subjected to noise. It is most commonly used as a measure for the reconstruction quality in 2D image compression. While the RMSE is the cumulative error between the reconstructed and the original mesh, PSNR is a measure of the peak error. A lower value for RMSE means smaller overall error, and as seen from the inverse relation between RMSE and PSNR, this translates to a high value of PSNR. Logically, a higher value of PSNR is good because it means that the ratio of signal to noise is higher. Here, the "signal" is the original mesh and the "noise" is the reconstruction error.

For the chicken as well as the dance sequence we outperform [LLWC08] in terms of quality and in terms of computation time. We achieve better PSNR for both sequences because the key-frames extracted by our framework are not necessarily frames from

Model	Vertices	Triangles	Frames
Chicken	3030	5664	400
Cow	2904	5804	204
Dance	7061	14118	201
Dolphin	6179	12337	101
Face	539	1042	10001

Table 5.2. Details of the sequences that were used in the section (see Figure 5.8).

Model	# KF	Our approach					GA approach [LLWC08]				
		Min. PSNR	Max. PSNR	Avg. PSNR	# Iter.	Time (sec.)	Min. PSNR	Max. PSNR	Avg. PSNR	# Gen.	Time (sec.)
Chicken	50	42.09	140.55	64.46	49	40	43.60	131.45	64.75	12238	1486
	40	38.44	102.36	60.25	46	27	39.75	127.04	59.35	9450	1249
	30	34.71	100.68	55.06	38	21	32.98	126.46	54.62	4578	711
	20	33.94	75.29	49.48	26	12	28.48	127.72	46.67	8180	1468
	10	29.88	67.96	41.30	27	10	21.58	135.80	40.06	2302	1123
Dance	50	52.56	78.54	65.65	46	55	49.89	69.30	62.14	5808	886
	40	50.14	73.76	60.89	42	39	46.27	65.45	56.99	9135	1390
	30	44.61	69.17	53.97	40	30	41.99	62.14	50.78	4414	1078
	20	38.96	57.01	47.54	42	19	36.65	54.71	43.41	4210	1855
	10	30.86	49.04	38.48	19	6	24.50	48.05	33.23	1702	547

Table 5.3. The PSNR error captures the logarithmically scaled ratio between signal and noise. Our approach extracts better key-frames and is 16 to 97 times faster for the dance sequence and 37 to 112 times faster for the chicken sequence.

the sequence but have a similar shape. Since they are the averages of the clusters they represent, we get better reconstructions of the whole sequence. The second important difference relates to the Genetic Algorithm (GA) used in [LLWC08], which needs a well-defined stopping criterion. Moreover, GAs converge very slowly because the mutation phase of any GA introduces a random element. Our BNG approach converges quadratically (see Section 5.2) and therefore yields better key-frames in a matter of seconds. Table 5.2 summarizes the statistics for the sequences used throughout the section. The timings in Table 5.3 were measured on an Intel Core2 Duo E6400 with 2GB of RAM.

5.2.4 Discussion

We introduced a fast and very simple pre-processing framework for animated meshes. It follows the general idea of PCA to extract the meaningful information from an an-

imation sequence as proposed in [AM00] and [KG04], but instead we use a recently introduced algorithm from machine learning for this task. PCA for an animated mesh is optimal in the 2-norm, but this optimality comes at a certain cost. The PCA eigenvectors cannot be directly interpreted as part of the sequence of meshes. Our framework clusters an animated mesh into frames representing similar motion. This gives us a certain set of frames similar to the PCA vectors, which we call key-frames. Though these frames are not part of the original mesh sequence, linear interpolation between them allows us to reconstruct the whole animation sequence. This approach introduces a small error compared to PCA, but the main advantage is that our key-frames reflect the motion inherent in the sequence much better, than the PCA vectors usually do.

Although the framework is already very efficient, there still exists some potential for improvement. The NG approach has recently been extended to the so-called *Patch Clustering* for streaming data [AH08]. This fits perfectly well for streaming compression, since it allows to compute the key-frames on the fly for a streaming sequence, thereby providing a valid clustering (key-frames) at any given time step.

5.3 Exploring Shape Space Alternatives

Until now, we have only considered the representation of a triangle mesh by its *3v vertex coordinates*, but we are not at all restricted to this *shape space*. For example, deformation gradients (see App. A) or the edge-angle representation described in Chapter 4.2 provide alternative possibilities. Within the following sections, we will explore the applicability of the aforementioned representations in terms of thinning mesh animations. The PCA method, as well as the NG approach, only requires the input data to be representable as a d-dimensional vector. For example, in case of deformation gradients, we could replace the mesh m_i in Eq. (5.1) by

$$m_i = \begin{pmatrix} \log(R_{i,1}) \\ S_{i,1} \\ \log(R_{i,2}) \\ S_{i,2} \\ \vdots \\ \log(R_{i,f}) \\ S_{i,f} \end{pmatrix} \in \mathbb{R}^{9f}. \tag{5.8}$$

Compared to the vertex-based approach described in Section 5.1 with $m_i \in \mathbb{R}^{3v}$, where v is the number of mesh vertices, we now have $m_i \in \mathbb{R}^{9f}$, where f is the number of faces (triangles) in the mesh.

In other words, we switched from the shape space of *vertex coordinates* $\mathfrak{S}_v := \mathbb{R}^{3v}$ to the shape space of *face coordinates* $\mathfrak{S}_f := \mathbb{R}^{9f}$. As demonstrated in Appendix A, a deformation gradient F_j for one triangle can be decomposed into the logarithm of the

rotational part $\log(R_j)$ and a scale/shear component S_j. Overall, this gives nine coefficients per triangle: three for the rotation and six for the scale-shear. As $f \approx 2v$ for triangle meshes by Euler's formula, this representation requires $9f \approx 18v$ coefficients compared to the $3v$ coefficients of the usual vertex representation, which in terms of reducing the amount of data seems to be counterproductive, yet we saw in Chapter 4 that taking linear combinations of these $9f$ face coordinates usually gives much nicer interpolated meshes than taking linear combinations of the $3v$ vertex coordinates. We even know of a third alternative. More precisely, the edge-angle representation, which is (at least in terms of interpolation) superior to face coordinates. Therefore, we consider for each of the e edges of the triangle mesh the corresponding edge length as well as the dihedral angle between the normals of the adjacent triangles and express the triangle mesh in terms of the shape space of *edge coordinates* $\mathfrak{S}_e := \mathbb{R}^{2e}$

$$m_i = \begin{pmatrix} s_{i,1} \\ \alpha_{i,1} \\ s_{i,2} \\ \alpha_{i,2} \\ \cdots \\ s_{i,e} \\ \alpha_{i,e} \end{pmatrix} \in \mathfrak{S}_e. \tag{5.9}$$

Since $e \approx 3v$ for a triangle mesh, again by Euler's formula, this shape space instance requires $2e \approx 6v$ coefficients, which is between vertex and face coordinates in terms of size. The remainder of this chapter focuses on the question which of the shape spaces, $\mathfrak{S}_v$, $\mathfrak{S}_f$ or $\mathfrak{S}_e$ allows for the best compact representation of an animation sequence.

5.3.1 PCA in Different Shape Spaces

Computing the PCA in the shape space of face coordinates $\mathfrak{S}_f$ or respectively in $\mathfrak{S}_e$, is only slightly more involved than computing it in $\mathfrak{S}_v$. Given a mesh $m_i \in \mathfrak{S}_v$ we already know how to express it in $\mathfrak{S}_f$ (see Eq. (5.8)) and in the same way as before, the PCA provides us with the eigenvectors for optimal reconstruction of this sequence.

Figure 5.9 summarizes the reconstruction results exemplarily for the handstand sequence, which consists of 175 meshes in total. The top left graph compares the root mean square error (RMSE) between the vertices of each reconstructed mesh and the corresponding original mesh from the sequence in relation to the number of PCA eigenvectors being used. Yet, this provides not necessarily a good measure for shape similarity as Figure 5.10 suggests. If we judge the reconstructed mesh in the middle of the figure on its own, the deviation from the original mesh on the left is not immediately visible, because it is visually not very distracting. The deficiency becomes more pronounced if we overlay both models as shown on the right of the figure. For example, if the mesh models extremities this problem is rather obvious. If just a few

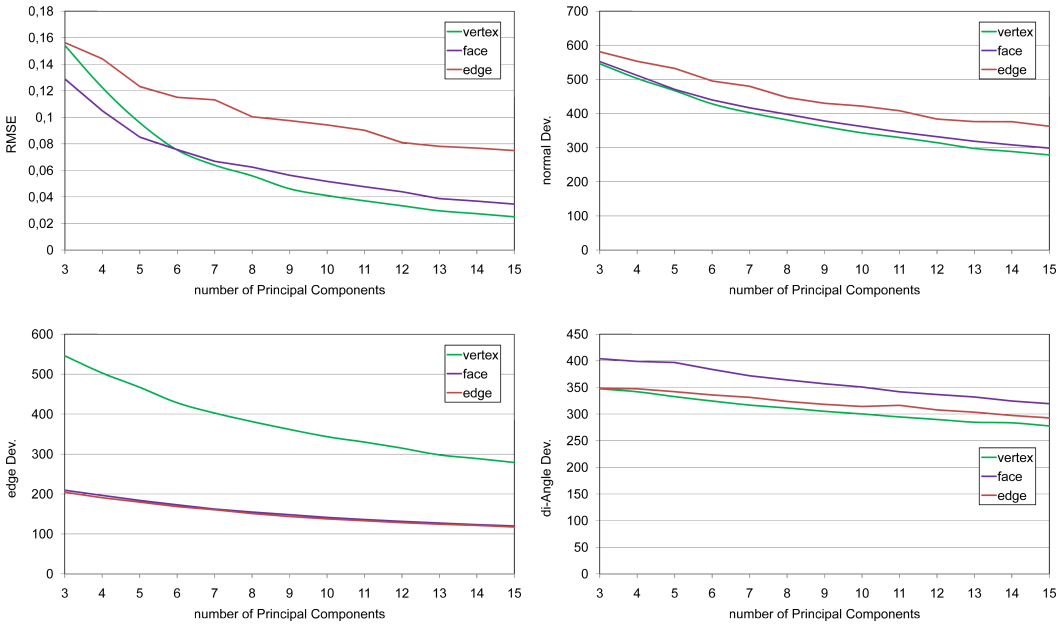

Figure 5.9. Reconstruction error for the handstand sequence (175 meshes) (Fig. 2.10, p. 17). Top left: RMSE between vertices. Top right: deviation of normals. Bottom left: deviation of edge-lengths. Bottom right: deviation of the dihedral angle between adjacent triangles.

triangles are slightly off, e.g. near the armpit, this causes the whole arm to deviate from the original model and thus causing a rather large RMSE. Yet visually, the error is very often only noticeable if the reconstructed mesh is compared directly to the original one. Obviously, this motivates to question the usability of the RMSE for measuring shape similarity.

Furthermore, please note that face coordinates are invariant to translation and edge coordinates moreover are independent of rotations, therefore we registered all meshes in the sequence against the first mesh, to allow for a fair comparison between the different shape spaces.

Therefore, we decided to additionally measure the average normal deviation for each face in the reconstructed mesh compared to the corresponding face in the original mesh (top right) in Figure 5.9. The bottom left graph shows the edge deviation, measured between the reconstructed mesh and the original and finally the bottom right graph in Figure 5.9 reflects the deviation in the dihedral angle, between the original mesh and its reconstruction. The top left graph in Figure 5.9 emphasizes the fact that using face coordinates, i.e. expressing the sequence in $\mathfrak{S}_f$, yields better reconstruction results than expressing it in $\mathfrak{S}_v$ for a small number of Principal Components. In terms of deviation of the normals (top right), face coordinates perform equally well, com-

Figure 5.10. Visualization of the root mean square error (RMSE) between the original mesh (left) and the reconstruction (middle) of an example mesh taken from the handstand sequence.

pared to vertex coordinates. This is not very astonishing, because the final stitching step involves the solution of a least-squares system (see App. A.3) which assures the required edge length in combination with a triangle orientation that leads to a smooth overall surface. This is also supported by the bottom left graph. Here, face coordinates and edge coordinates perform equally well. The visible difference in the bottom right graph is caused by the same effect. Due to the aforementioned least-squares system, face coordinates usually produce a smoother surface than edge coordinates.

Although these results seem quite promising, we should keep in mind that we still have to deal with the introduced overhead in $\mathfrak{S}_f$ and $\mathfrak{S}_e$. The major difference compared to the PCA being executed in $\mathfrak{S}_v$ is that obviously the eigenvectors now consist of the combination of logarithms $\log(R_j)$ and scale-shear components S_j, for each F_j, because they have to be in the same format as the data points in the corresponding shape space (see Eq. (5.8)). We could simply store them in their current format, but we already know that in doing so we would introduce a significant amount of overhead. Remember, for deformation gradients we need to store $18v$ values instead of $3v$, respectively $6v$ in case of $\mathfrak{S}_e$. Furthermore, the PCA eigenvectors in their current format do not reflect meshes from the sequence and thus no longer present suitable input to the compression methods mentioned in Section 2.4.

Fortunately, this issue can easily be resolved, because the representation of the PCA eigenvectors, either in $\mathfrak{S}_f$ as well as in $\mathfrak{S}_e$, is very similar to the interpolation results in the corresponding shape space. Thus, we are able to *up-* and *downcast* the PCA eigenvectors from their particular shape space instantiation into $\mathbb{R}^d$ with the proper tools presented in Chapter 4.2 and Appendix A.3. For example, for each given Principal Component, either in $\mathfrak{S}_f$ or $\mathfrak{S}_e$, upcasting it into a mesh would involve the following steps:

- add the mean back as shown in section 5.1.1

	Format	handstand	walk2
direct	TXT	627 kB	130 kB
	ZIP	262 kB	57 kB
	MSGI	235 kB	51 kB
upcast	PLY	489 kB	106 kB
	OpenCTM	130 kB	27 kB

Table 5.4. File sizes of different formats for storing the $\mathfrak{S}_e$ PCA eigenvectors of the handstand and the walk sequence (courtesy of [BVGP09]). The upper three rows reflect the encoding of the PCs as plain text, zipped and binary encoded, while the lower rows show the size after upcasting the PCs into $\mathbb{R}^d$ as described in Section 5.3.1.

- use the appropriate reconstruction from $\mathfrak{S}_f$ or $\mathfrak{S}_e$ to $\mathbb{R}^d$
- store the mesh in a compressed format

In Table 5.4 we list the space requirements for different ways of storing the Principal Components. The first three rows consider the storage in the current shape space format, either as plain text, internal binary or zipped file. The lower two rows reflect the upcasted representations, either as uncompressed binary ply file or compressed in the OpenCTM [Gee09] format. Obviously, upcasting the eigenvectors from the actual shape space into $\mathbb{R}^d$ provides a significant advantage, if a reasonable compression method is used. In this case we used OpenCTM, because it is freely available, very easy to integrate and due to the implemented stat-of-the-art entropy-reduction/coding schemes it produces only slightly worse results (in terms of size) than for example the benchmark coder presented by Isenburg et al. [IAS02].

Yet, up- and downcasting introduces a major drawback to the thinning approach as emphasized by Figure 5.11. Independent of the shape space in which the error is measured, it is increased significantly, if the up- and downcasting step is involved. This is due to the fact that the Principal Components are no longer guaranteed to be orthogonal to each other in their $\mathbb{R}^d$ representation, which is a crucial requirement of the PCA.

As a possible solution to counteract this misery, we could postpone the computation of the weights for the linear combinations of the Principal Components. Therefore, we simply have to alter the scheme introduced in Section 5.1.1 in the following way:

- set up the matrix M from the sequence $\boldsymbol{m}$
- subtract the sequence mean (Eq. (5.3)) from each column of M
- compute the Principal Components

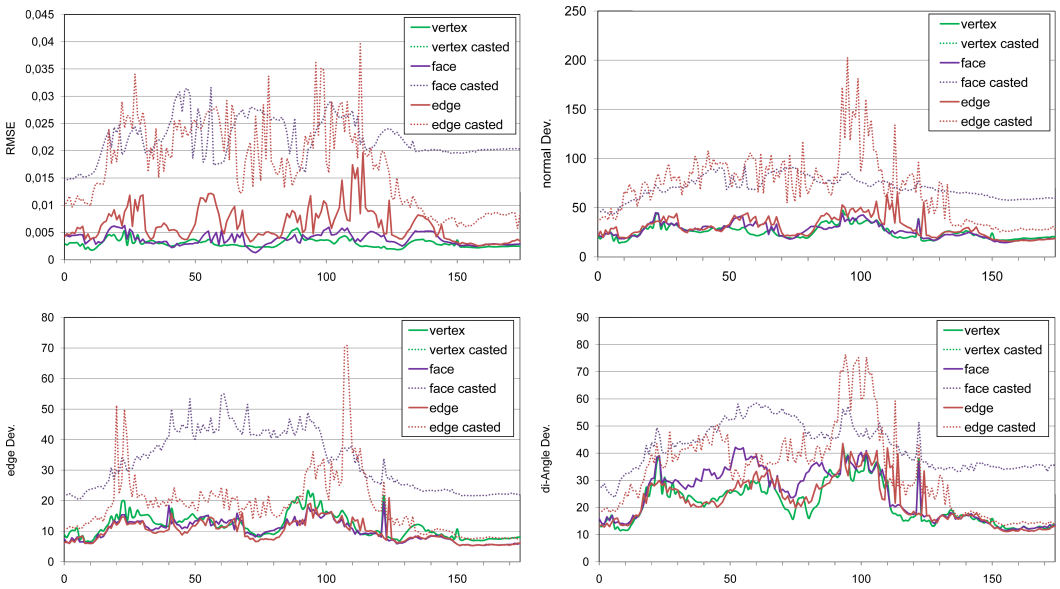

Figure 5.11. Reconstruction error for the PCA approach for each mesh in the handstand sequence (Fig. 2.10, p. 17), with 10 Principal Components. Top left: RMS error between vertices. Top right: deviation of normals. Bottom left: deviation of edge-lengths. Bottom right: deviation of the dihedral angle between adjacent triangles. As soon as up-/downcasting of the Principal Components is involved, the error increases significantly, because the PCs are no longer orthogonal.

- add back the mean (Eq. (5.3)) to each Principal Component and upcast it to $\mathbb{R}^d$
- solve a least-squares system to find the optimal interpolation weights for the upcasted Principal Components

But this approach lacks any theoretical foundation and a direct comparison shows that the NG approach performs better in this setting, as shown in Figure 5.12. Although, as we already know from Section 5.2 the NG approach usually yields slightly worse results than the PCA approach, it can adapt much better to this situation and it provides much more robustness, especially if a lossy compression scheme is used for compressing the Principal Components (respectively the NG key-frames), because this would also strongly influence the vertex positions.

5.3.2 Discussion

Computing the PCA in $\mathfrak{S}_v$ for an animated mesh is optimal in the 2-norm, but this optimality comes at a certain cost. The PCA eigenvectors cannot be directly interpreted as part of the sequence of meshes. Yet, there is a simple solution, by adding the average

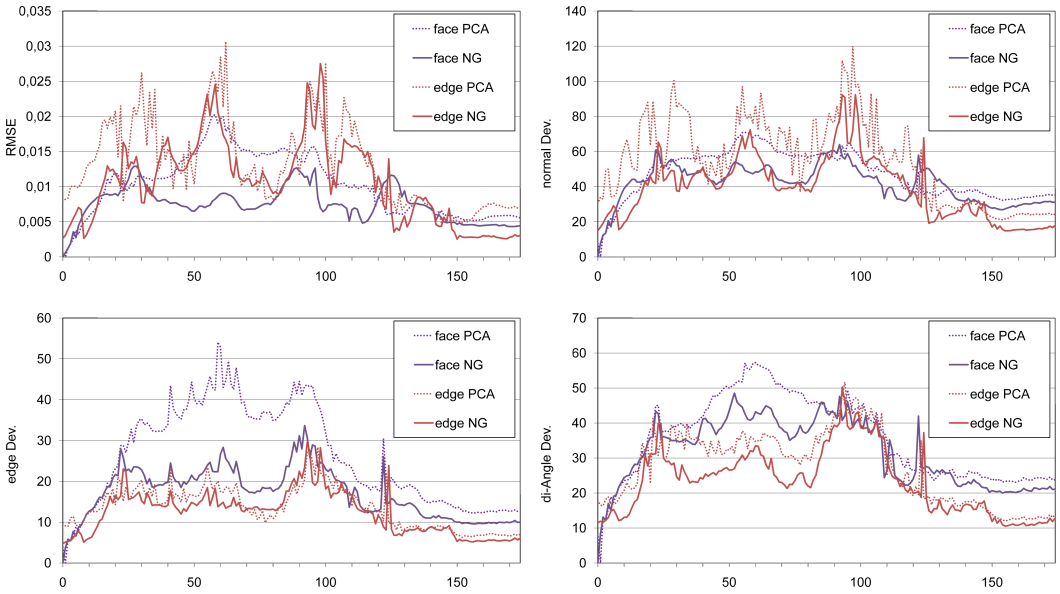

Figure 5.12. Comparison between the NG and the PCA approach. This variant of the PCA circumvents the problem of the non-orthogonal Principal Components by computing the interpolation weights after the upcasting step.

(mean) mesh of the sequence to each computed Principal Component as shown in Section 5.1.1. Based on the results from Chapter 4, we know that linear combinations of face and edge coordinates usually yield much nicer interpolation results.

Therefore, we explored different shape spaces to see whether the animation sequence could be better embedded in these spaces. Unfortunately, the shape spaces introduce some unwanted overhead to our mesh representation. To annihilate this overhead, we applied the same mechanisms that we used in Chapter 4 to reconstruct the meshes after interpolation, since the representation of the Principal Components in $\mathfrak{S}_f$ and $\mathfrak{S}_e$ is very similar to the interpolation result for the specific method.

This justified the implementation of an up-/downcasting step from the particular shape space into $\mathbb{R}^d$, such that we could benefit from applying an appropriate mesh compression scheme (see Tab. 5.4). Yet, casting the meshes between the corresponding shape space and $\mathbb{R}^d$ reveals a major drawback of this approach. After upcasting the Principal Components they are not necessarily orthogonal to each other any longer, which is an essential requirement for the PCA to produce correct results.

A possible way to avoid this pitfall is to postpone the computation of the weights for the linear combinations to a later stage in the thinning. Yet, in Section 5.2 we introduced a fast and very simple pre-processing framework for animated meshes that could be used instead. Besides the aforementioned problem of non-orthogonal Principal Components, there is another significant difference compared to vertex coordinates. For edge coordinates we depend on two major requirements. Firstly, the angle

Figure 5.13. Edge space requirements vs. PCA. *Left*: the first mesh in a short walking sequence of a blender toy example. *Middle*: a Principal Component casted from $\mathfrak{S}_f$ into $\mathbb{R}^d$. *Right*: the same PC, but casted from $\mathfrak{S}_e$ into $\mathbb{R}^d$. In regions with small and skinny triangles, neither the PCA nor the NG approach necessarily comply with the requirements of the edge shape space.

should be in the range of $[-\pi, \pi]$ and secondly, the interpolated edge lengths should always yield a triangle. In terms of linear interpolation of triangles we can guarantee that these requirements are always fulfilled, but neither the PCA nor the NG approach accounts in any form for these restrictions.

Thus, it could happen that the PCA reveals eigenvectors (respectively NG key-frames) which violate these restrictions on angles and edge lengths. If we further cast these eigenvectors into meshes, this leads to tremendous failures in the gluing hierarchy of edge coordinates (see Sec. 4.2.2 and Sec. 4.2.3) which is best illustrated by the toy example in Figure 5.13. The left picture shows the first mesh in a walking sequence of a simple Blender model. In the middle of the figure, one of the Principal Components was converted from $\mathfrak{S}_f$ to $\mathbb{R}^d$ and on the right the same Principal Component was converted from $\mathfrak{S}_e$. Especially in those areas of the mesh where many long and skinny triangles are needed to model a smooth transition between different parts of the mesh (e.g. at the hip), a small deviation from the requirements could already cause a dramatic effect in the reconstruction of the Principal Component, respectively the NG key-frame in the shape space of edges. In our experiments we circumvented these restrictions by considering $\tan(\frac{\alpha}{2}) \in \mathbb{R}$ for the dihedral angle α and $\log(l) \in \mathbb{R}$ for the edge length l instead of their direct representations (α in the range $[-\pi, \pi]$, respectively $l \in \mathbb{R}^+$). However, the improvement was insignificant, because this trick still does not guarantee the scaled edges to close up and yield a triangle.

Although applying the Mesh Massage framework (see Sec. 3) allows us to significantly reduce the said effect for edge coordinates (see Fig. 5.14), we are not able to overcome it completely. The situation for the face coordinates is similar, yet they are much less affected, because of their different encoding and due to the final global optimization step of the least-squares system. For face coordinates, the optimization

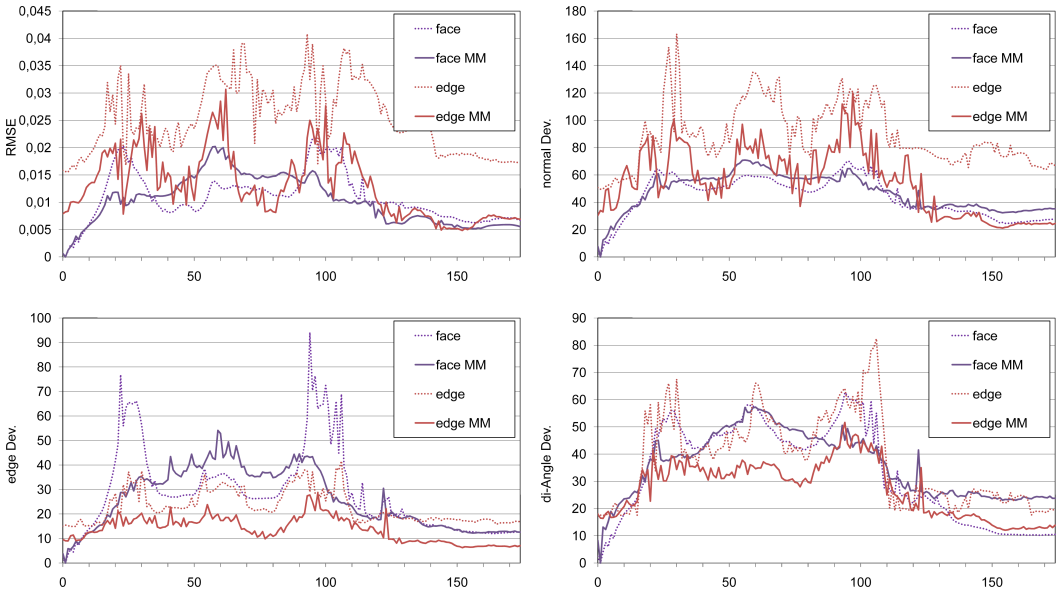

Figure 5.14. For each error, we compare the results between the non-modified sequence and a modified one. For the modified sequence, we applied the Mesh Massage Framework (see Sec. 3) to "copy&paste" the shape of the triangles from the first mesh of the sequence, to the consecutive meshes.

	face	face MM	edge	edge MM
RMSE	0.154752	0.147371	0.344291	0.189548
normal dev.	610.154	646.159	1219.48	790.322
edge dev.	441.69	381.198	315.108	193.274
angle dev.	496.427	496.967	544.122	386.981

Table 5.5. The average errors for the complete sequence, with and without optimization through the Mesh Massage framework. While the error in $\mathfrak{S}_f$ stays more or less constant, the influence of the optimization is significant in $\mathfrak{S}_e$.

merely distributes the error slightly differently in the sequence, as we can see from Figure 5.14 and Table 5.5.

While there is only little change in the face coordinates, the improvement in the edge coordinates is quite remarkable, since they are lacking a final global optimization step, which takes all edges into account (compared to the face coordinates). This suggests that neither the PCA nor the NG approach are the appropriate methods for reducing the redundancy in the shape space in a proper way. Overall, the aforementioned difficulties lead to a significantly higher reconstruction error if up-/downcasting is involved. This more or less consumes the benefits we gain through the usage of edge

and face coordinates. Therefore, a compact representation of an animation sequence in different shape spaces other than $\mathfrak{S}_v$ with the presented methods is not advisable.

Chapter 6

Conclusion and Future Work

Within this thesis we pictured the layout of a possible geometry processing pipeline, ranging from shape acquisition, surface reconstruction, optimization to compact representations. Due to the methods utilized for shape acquisition and surface reconstructions nowadays, the resulting triangle meshes very often allow or even require further optimization. At a first glance the fact, that the real object is only modeled by a set of triangles, each being a piecewise linear approximation to the real surface, is a disadvantage. But, actually this facilitates the modification and optimization of triangle meshes, since it bears the freedom for the surface samples (the mesh vertices) to freely move in space, as long as they do not lose their property of being surface samples.

We presented a simple and fast method for optimizing triangle meshes in Chapter 3 and showed that it provides enough flexibility to be applied to other applications beyond the optimization of a single static mesh. The flexibility stems from the fact that we concurrently optimize the shape of the triangles and the Hausdorff distance to an arbitrary reference geometry by minimizing a quadratic energy. We have thoroughly compared our framework to the competing approaches of Nealen et al. [NISA06] and Liu et al. [LTJW07].

One of the major advantages of our method is that during the optimization, the features of the mesh are nicely preserved, because the approximate Hausdorff distance always pulls the vertices close to the feature lines. But apart from this constraint, all vertices are free to move and acquire the desired triangle shapes. This course of action covers exactly all possibilities that we have in modifying a triangles mesh, as stated previously. The mesh being optimized is basically free to “float” over the reference geometry, similar to a tight suit or a second skin. Yet, a potential drawback of our framework is the calculation of the corresponding points. If the distance between the meshes is too large, they cannot be determined correctly anymore.

If we follow the thought of a tight second layer floating over the reference geometry, then another application becomes immediately visible, given a set of triangle meshes –scanned and processed independently– our framework is able to convert this

set into a new one that consists of compatible meshes instead. As one of the example applications presented in Chapter 4 we successfully employed our framework on exactly that particular setting, since generating motion out of a few meshes is mainly the topic of said chapter. At the beginning we surveyed existing techniques, such as deformation gradients introduced by Sumner et al. [SP04; SZGP05; Sum06]. For interpolation purposes and mesh modeling in general, they have proven to be a valuable tool. This is mainly due to their simplicity and robustness in the implementation, but since their computation relies on a reference mesh or more precisely, a reference triangle, they can sometimes produce counterintuitive results, if used for interpolation of large rotations. As stated in the thesis of Sumner [Sum06] it is possible to fix this non-intuitive behavior, yet "whether proper corrections can be found for all cases is an open problem." Therefore, we give a detailed analysis at the beginning of Chapter 4, why deformation gradients should not necessarily be the first choice for interpolation in the presence of large rotations.

Furthermore, we presented a novel method for interpolating between two or more compatible meshes, which is not based on local affine transformations. Instead we linearly interpolate the intrinsic local properties of the input meshes, which is the most natural and simplest thing to do on the level of wedges (two adjacent triangles). The simplicity of this idea is counterweighted by the fact that putting the wedges together such that they yield a globally consistent mesh is a rather complex optimization problem. However, we showed that this problem can be solved in principle by multi-registration methods and since this can be combined with a hierarchical decomposition of the mesh in larger and larger patches, it can actually be solved efficiently as well.

Our approach yields very intuitive interpolations and can hence be used for exploring the natural shape space spanned by a set of key poses. This allows for a number of interesting applications. For example, approaches like the one of Smith et al. [SPK+07] for shape space exploration of auto parts, our presented approach for sequence thinning [WDH+08] or the method propose by Hasler et al. [HSS+09] for learning a statistical model of body poses can benefit from our method. Furthermore, it can also be utilized for shape editing based on user-defined constraints, similar to how it is described in [KMP07]. Moreover, our framework can be used to express animation sequences as simple 2D paths in a reference polygon (compare Figure 4.21), which in turn may provide an intuitive tool for character animation or even crowd generation.

In Chapter 5 we explored how to find these few key poses, in other words how to purge the redundant information from a given animation sequence. We recapitulated how to apply the standard tool for dimensional reduction (PCA) to an animation sequence, as well as we introduced a fast and very simple pre-processing framework, that follows the general idea of PCA, but instead uses a recently introduced algorithm from machine learning for this task.

Furthermore, inspired by the results from Chapter 4 we introduced the concept of different shape spaces for triangle meshes. The most common way to represent a

triangle mesh is by its vertex coordinates, but as we have learned from deformation gradients, it is also justifiable to express the mesh in terms of rotation and scale/shear components per face, which motivates to speak of face coordinates in this case. A third opportunity for a shape space is provided by the edge/angle representation, based on wedges, that we introduced in Chapter 4. For obvious reasons, in this case we speak of the shape space of edge coordinates.

The remainder of Chapter 5 considered the question, which of the aforementioned shape spaces is optimal for a compact representation of an animation sequence. Unfortunately, the shape spaces other than the vertex space introduce quite a significant overhead in their mesh representations, but the hypothesis based on the gained experiences from the interpolation chapter was that a sequence can be represented with fewer key frames in these shape spaces. Which is actually confirmed by our results, but to avoid the overhead of the particular shape spaces, we had to upcast the meshes from their current shape space representation into $\mathbb{R}^d$, because this allows for the application of existing mesh compression methods. Yet, we had to realize that this upcasting slightly changes the Principal Components, such that they are no longer orthogonal and thus breaking the method. Although we were able to show that the NG approach provides a valid alternative solution in this case, there still exists a major drawback, due to the requirements the edge coordinates have.

More precisely, edge coordinates require the angle α to be within the range $[-\pi, \pi]$ and that the scaled edges can be assembled to a triangle. Neither the PCA approach, nor the NG approach "knows" of these requirements. Therefore, it may happen that the linear combinations of Principal Components (respectively key-frames) violate these restrictions, especially in the presence of long and skinny triangles, causing flipped triangles in the worst case. And since the conversion step from $\mathfrak{S}_f$ to $\mathbb{R}^d$ relies on the assumption that the wedges (and the following larger patches in the hierarchy) have a common overlap, this may lead to a rather large error, if the Principal Components (respectively key-frames) are upcasted into a meshes. In our experiments we circumvented these restrictions by considering $\tan(\frac{\alpha}{2}) \in \mathbb{R}$ for the dihedral angle and $\log(l) \in \mathbb{R}$ for the edge lengths instead of their direct representations (α in the range $[-\pi, \pi]$, respectively $l \in \mathbb{R}^+$). However, the improvement was insignificant, because this still does not guarantee that the scaled edges yield a triangle.

This suggests that although PCA and Neural Gas are able to reduce the redundant information in a given animation sequence, they are not custom-tailored for this purpose in other shape spaces than $\mathfrak{S}_v$. Therefore, future research should incorporate the search for better methods to reflect the manifold of meshes in the corresponding shape spaces.

Yet, the basic idea of representing motion by a reduced set of carefully selected frames and expressing the whole sequence as a path in a lower dimensional space (see Fig. 4.21), still seems promising and allows for a variety of possible applications, for example given a valid interpolation path in the lower dimensional space, we could

easily replace the key-frames and thus simply “copy&paste” the motion onto another object with possible applications such as crowd generation or shape space design as shown by Smith et al. [SPK+07]. In this context we have basically three degrees of freedom, we could modify the interpolation curve in the lower dimensional space, we could modify the shapes as previously mentioned, or we could modify the vertices that are associated with the meshes in the lower dimensional space.

Appendix A

Deformation Gradients

Deformation gradients were first introduced by Barr [Bar84] in the context of continuum mechanics. In this area they provide a tool to model the deformation of an object when it is under heavy load. Sumner et al. [SP04] were the first to utilize this representation to capture the deformation of a 3D mesh and later used it in [SZGP05] for interpolation among several meshes. Because of their versatility deformation gradients became quite popular over the recent years. An extensive explanation can be found in the Ph.D. Thesis of Sumner [Sum06]. This appendix briefly summarizes the steps necessary to use deformation gradients as a tool for interpolation between two or more triangle meshes.

For a triangle mesh the gradient of the continuous deformation function can be discretized as a piecewise constant tensor field with one constant tensor $J \in \mathbb{R}^{3\times 3}$ for each triangle [Sum06]. These *deformation gradients* equal the affine transformation that converts the independent edge vectors of the standard 3-simplex into the spanning vectors (and the scaled normal) of the current triangle.

At first we clarify what a standard 3-simplex is. A n-dimensional simplex Δ^n can be defined as the convex hull of its $n+1$ vertices. Thus, the regular 3-simplex is equivalent

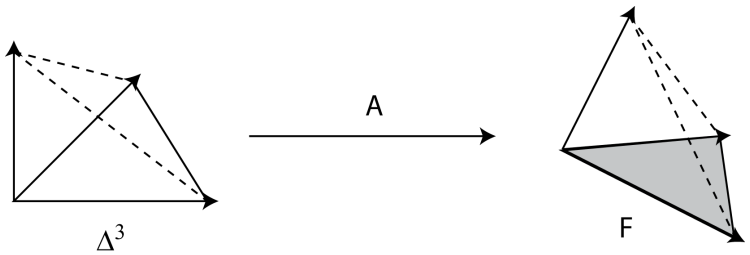

Figure A.1. The deformed frame F is created by applying the deformation gradient A to the regular 3-simplex Δ^3.

to the tetrahedron with vertices $s_1 = (1,0,0)$, $s_2 = (0,1,0)$, $s_3 = (0,0,1)$, $s_4 = (0,0,0)$. This provides us with an local orthonormal frame with edges $e_i = s_i - s_4 = s_i$ for $i = 1,\ldots,3$. Given a random triangle with vertices $v_i \in \mathbb{R}^3$ for $i = 1,\ldots,3$, we can turn it into a second tetrahedron (or frame) F by additionally computing a (scaled) normal $n = (v_2 - v_1) \times (v_3 - v_1)$.

If we think of the standard 3-simplex as an undeformed frame and the second tetrahedron as the same frame but deformed, we are interested in the linear transformation which maps Δ^3 to F. It is rather obvious that this transformation is given by

$$\phi(x) = Ax + v_1 \qquad \text{with } x \in \mathbb{R}^3.$$

We could now use $\phi(x)$ to interpolate between Δ^3 and F. For a single triangle this is maybe appropriate. But if we applied this scheme to each triangle of a mesh, it would produce a triangle soup, caused by the translational component. Therefore, we postpone the question where to position the triangle in space for a moment. Because we are only interested in the orientation and shape of the tetrahedron we focus instead on the gradient of $\phi(x)$

$$\nabla(\phi) = A$$

the *deformation gradient*. In general, the triangle from which the reference frame F is constructed, can be chosen arbitrarily. We compute the deformation gradient between a frame F_i with $i = 1,\ldots,n$ and the reference frame F as

$$J_i = A_i A^{-1}. \tag{A.1}$$

In case we are using the regular 3-simplex as the reference frame, this is a special case (see Fig. A.2), because $\Delta^3 = Id$ and it is not necessary to compute A explicitly, since $A = J$ in this case. Now that we know how to model the deformation of one triangle, it is feasible to use this knowledge for interpolation between two triangles. The main idea is to compute the deformation gradients J_0 and J_1 for two triangles and apply the interpolation result $J_t = (1-t)J_0 + tJ_1$ to the reference frame, which then yields the interpolated triangle. This method is depicted in Figure A.2.

Please note that the choice of the reference frame is more or less arbitrary. We can either use the regular 3-simplex Δ^3 or an already deformed frame F, computed from a triangle of a reference mesh. Both ways are sketched in Figure A.2.

Hence, interpolation of deformation gradients is a bit more subtle, we have to take care of the fact that an affine transformation can consist of rotation, scaling and shearing. Linear interpolation of the latter two poses no problem, but the challenge lies in the rotational part, due to its non-Euclidian nature. Sumner et al. [SZGP05] therefore propose to split the deformation gradients into a rotational part R and a scale/shear part S and interpolate them independently using the matrix logarithm and exponential for the rotations. The remainder of this chapter is therefore organized

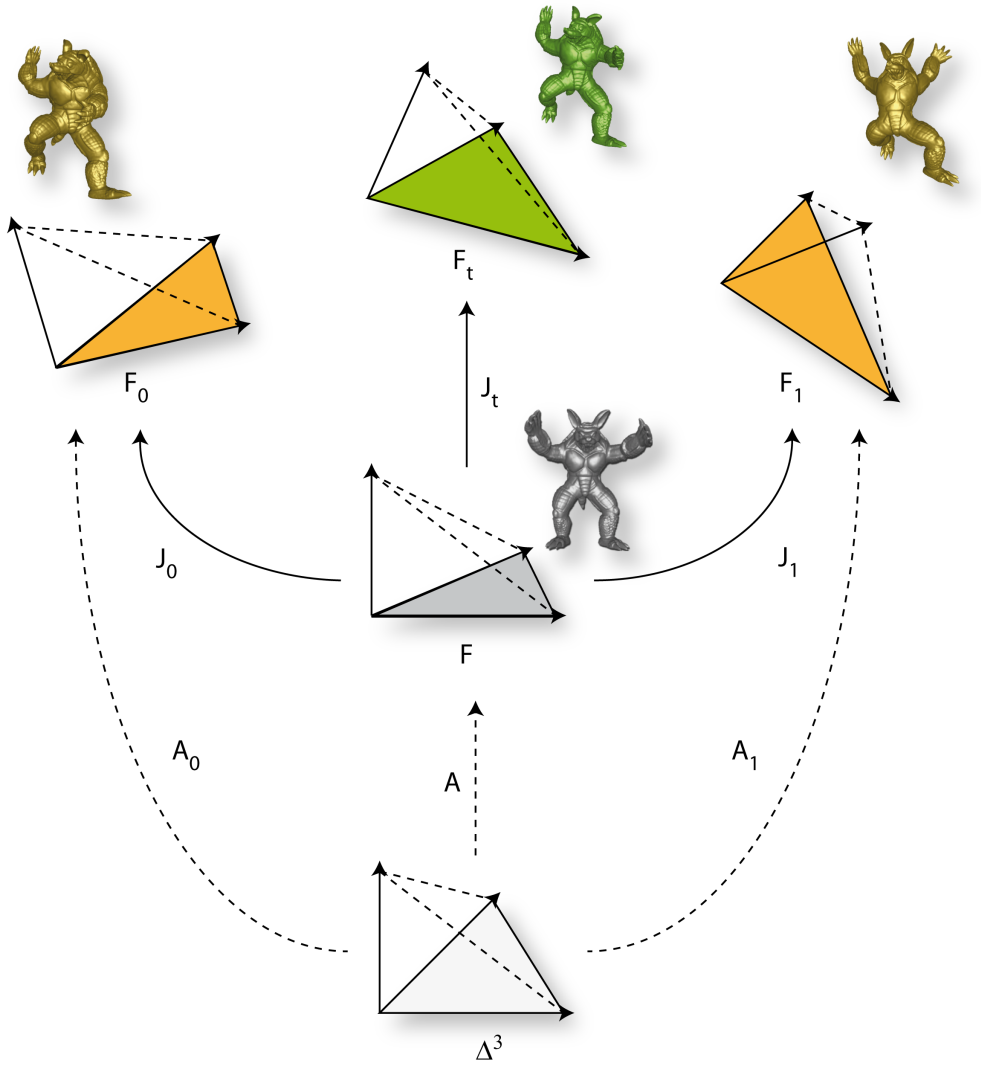

Figure A.2. Interpolation of deformation gradients. Given the local frames F_0 and F_1, we find the corresponding deformation gradients by using Eq. (A.1). If the interpolated gradient, computed as $J_t = (1-t)J_0 + tJ_1$, is now applied to the reference frame F it will produce the interpolated frame F_t.

as follows. In section A.1 we take a look on how to properly decompose deformation gradients into their said components. Section A.2 will demonstrate how to compute the matrix exponentials and logarithms for a given rotation matrix in 2D as well as in 3D. Finally, we combine the results from those chapters and show how to use deformation gradients for interpolation as proposed by Sumner et al. [SZGP05] in section A.3.

A.1 Deformation Gradient Decomposition

Given a deformation gradient J we are looking for the rotation R and the scale/shear component S of that matrix such that

$$J = RS.$$

Shoemake and Duff [SD92] analyze the best way for such a decomposition and promote to apply a polar decomposition on the deformation gradient J to get the rotation R and the scale/shear component S.

They state that a polar decomposition can be computed by using the results of a singular value decomposition (SVD), but due to the fact that this is computational intense, they provide the following alternative. For a given deformation gradient J we want to find a rotation matrix R, such that the distance

$$||R - J||_{\mathbf{F}}$$

measured in the Frobenius norm is minimal. This can be implemented as shown by Higham [Hig86]

$$\begin{aligned} &R_0 = J \\ &\text{repeat} \\ &\quad R_{i+1} = \tfrac{1}{2}\left(R_i + R_i^{-\mathbf{T}}\right) \\ &\text{until } R_{i+1} - R_i \approx 0. \end{aligned} \tag{A.2}$$

Now that we have extracted the rotation R from J, the scale/shear part can easily be computed as $S = R^{-1}J$. By exploiting the property of rotation matrices that $R^{-1} = R^T$ this can be simplified to $S = R^T J$.

A.2 Matrix Exponentials

Sumner et al. [SZGP05] suggest to change the representation of the rotation matrices to the angle-axis representation. Therefore they use the so called exponential map which is a surjective mapping form the Lie algebra of skew-symmetric matrices $\mathfrak{so}(n)$ to the special orthogonal group $SO(n)$ formed by rotation matrices (e.g. see [ML78; GX02; Ale02a; ML03]). A complete coverage is well beyond the scope of this chapter, we will therefore focus on the 2D as well as on the 3D case.

A.2.1 Matrix Exponential in 2D

In 2D the rotation matrix

$$R = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

describes the rotation in the xy-plane with the angle θ and we can obviously rearrange it in the following way

$$R = I \cdot \cos\theta + B \cdot \sin\theta \tag{A.3}$$

with $I = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$ and $B = \left(\begin{smallmatrix} 0 & -1 \\ 1 & 0 \end{smallmatrix}\right)$.

We can further replace $\cos\theta$ and $\sin\theta$ in Eq. (A.3) by their Taylor expansions

$$\begin{aligned} R &= I \cdot \underbrace{\left(1 - \tfrac{\theta^2}{2!} + \tfrac{\theta^4}{4!} - \tfrac{\theta^6}{6!} + \ldots\right)}_{\cos\theta} + B \cdot \underbrace{\left(\theta - \tfrac{\theta^3}{3!} + \tfrac{\theta^5}{5!} - \tfrac{\theta^7}{7!} + \ldots\right)}_{\sin\theta} \\ &= I - I\tfrac{\theta^2}{2!} + I\tfrac{\theta^4}{4!} - I\tfrac{\theta^6}{6!} + \ldots + B \cdot \left(\theta - \tfrac{\theta^3}{3!} + \tfrac{\theta^5}{5!} - \tfrac{\theta^7}{7!} + \ldots\right) \end{aligned} \tag{A.4}$$

If we take a look at the power series of B, we realize that since $B^0 = I$ and $B^2 = -I$ it follows that $B^{(2k)} = (-1)^k I$ for each $k \in \mathbb{N}$ and therefore Eq. (A.4) can further be written as

$$\begin{aligned} R &= I + B\theta + \tfrac{(B\theta)^2}{2!} + \tfrac{(B\theta)^3}{3!} + \tfrac{(B\theta)^4}{4!} + \tfrac{(B\theta)^5}{5!} + \tfrac{(B\theta)^6}{6!} + \ldots \\ &= \sum_k \frac{(B\theta)^k}{k!} \\ &= e^{B\theta}. \end{aligned}$$

Applying the logarithm on both sides leads to $\log(R) = B\theta$ and motivates to speak of the *skew-symmetric* matrix B scaled by the angle θ as the *matrix logarithm* of the *rotation matrix* R. The matrix *exponential* and *logarithm* are often applied to the task of motion interpolation (e.g. see [SD92; KKS95; GX02; Ale02a]).

A.2.2 Matrix Exponential in 3D

In 3D the general rotation matrix for rotating about an arbitrary normalized vector $e = (e_1, e_2, e_3)$ with angle θ is of the following form

$$R = \begin{pmatrix} \cos\theta + e_1^2(1-\cos\theta) & e_1e_2(1-\cos\theta) - e_3\sin\theta & e_1e_3(1-\cos\theta) + e_2\sin\theta \\ e_2e_1(1-\cos\theta) + e_3\sin\theta & \cos\theta + e_2^2(1-\cos\theta) & e_2e_3(1-\cos\theta) - e_1\sin\theta \\ e_3e_1(1-\cos\theta) - e_2\sin\theta & e_3e_2(1-\cos\theta) + e_1\sin\theta & \cos\theta + e_3^2(1-\cos\theta) \end{pmatrix}.$$

The rotation angle θ can be extracted from this matrix by calculating the trace

$$\begin{aligned} \text{trace}(R) &= \cos\theta + e_1^2(1-\cos\theta) + \cos\theta + e_2^2(1-\cos\theta) \\ &\quad + \cos\theta + e_3^2(1-\cos\theta) \\ &= 3\cos\theta + (e_1^2 + e_2^2 + e_3^2)(1-\cos\theta) \\ &= 1 + 2\cos\theta \end{aligned} \tag{A.5}$$

where the last simplification is due to the fact that e is of unit length. Since R is known in our case, we can rearrange the last line slightly, such that

$$\theta = \arccos\left(\frac{\text{trace}(R)-1}{2}\right) \tag{A.6}$$

and extract the rotation angle θ this way (see [GX02]). The next step is to extract the rotation axis, Gallier et al. [GX02] observe that

$$\begin{aligned} R - R^{\mathrm{T}} &= \begin{pmatrix} 0 & -2e_3\sin\theta & 2e_2\sin\theta \\ 2e_3\sin\theta & 0 & -2e_1\sin\theta \\ -2e_2\sin\theta & 2e_1\sin\theta & 0 \end{pmatrix} \\ &= \underbrace{\begin{pmatrix} 0 & -e_3 & e_2 \\ e_3 & 0 & -e_1 \\ -e_2 & e_1 & 0 \end{pmatrix}}_{=:\,B} \frac{1}{2\sin\theta} \end{aligned} \tag{A.7}$$

which reveals the rotation axis (our normalized vector $e = (e_1, e_2, e_3)$) as the entries of the skew-symmetric matrix B.

For similar reasons as in the 2D case this motivates to speak of $B\theta$ as the logarithm of R,

$$\begin{aligned} R &= e^{B\theta} \\ &= \sum_k \frac{(B\theta)^k}{k!} \\ &= I + B\theta + \tfrac{B\theta^2}{2!} + \tfrac{B\theta^3}{3!} + \ldots \end{aligned} \tag{A.8}$$

if we examine the powers of B, we see that $B^0 = I, B^1 = B, B^2 = B^2, B^3 = -B, B^4 = -B^2$, and $B^{k+4} = B^k$ for $k \geq 1$, so we can replace the appropriate powers and express Eq. (A.8) as

$$\begin{aligned} &= I + B(I\theta - I\tfrac{\theta^3}{3!} + I\tfrac{\theta^5}{5!} - \ldots) + B^2(I\tfrac{\theta^2}{2!} - I\tfrac{\theta^4}{4!} + I\tfrac{\theta^6}{6!} - \ldots) \\ &= I + B\cdot I\underbrace{(\theta - \tfrac{\theta^3}{3!} + \tfrac{\theta^5}{5!} - \ldots)}_{\sin\theta} + B^2\cdot I\underbrace{(\tfrac{\theta^2}{2!} - \tfrac{\theta^4}{4!} + \tfrac{\theta^6}{6!} - \ldots)}_{1-\cos\theta} \\ &= I + B\sin\theta + B^2(1-\cos\theta) \end{aligned} \tag{A.9}$$

which is the well known *Rodrigues Rotation Formula* [GX02] for converting the axis-angle representation into a rotation matrix.

However, one final remark concerning Eq. (A.7) has to be made. It is only valid for $\sin\alpha \neq 0$, so we have to take care of two cases. The first case, $\alpha = 0$ corresponds obviously to no rotation at all, the logarithm in this case is defined as

$$B = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

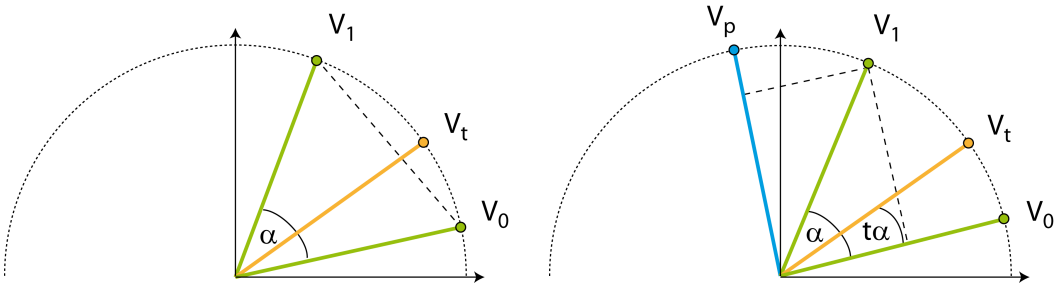

Figure A.3. Interpolation of Rotation Axes. Left: normalized linear interpolation between V_0 and V_1. Right: spherical linear interpolation.

which leads to R being the identity matrix, since $R = e^{B\cdot\theta} = I$. In the second case, $\alpha = \pi$ and the general rotation matrix is of the form

$$R = \begin{pmatrix} 2e_1^2 - 1 & 2e_1e_2 & 2e_1e_3 \\ 2e_1e_2 & 2e_2^2 - 1 & 2e_2e_3 \\ 2e_1e_3 & 2e_2e_3 & 2e_3^2 - 1 \end{pmatrix}.$$

Due to this simple form, the rotation axis can be extracted easily from the diagonal. Therefore, in case of $\alpha = \pi$ we have

$$\log(R) = B \cdot \pi = \begin{pmatrix} 0 & -e_3 & e_2 \\ e_3 & 0 & -e_1 \\ -e_2 & e_1 & 0 \end{pmatrix} \cdot \pi.$$

A.2.3 Lerp, Nlerp, Slerp?

Although the axis-angle representation is much easier to handle in terms of interpolation, there still are some minor difficulties we have to deal with.

Given two rotation axis, extracted as normalized vectors from some rotation matrices, the remaining question is, how to interpolate between them or in other words, how to find V_t in Figure A.3. Linear interpolation, also known as *Lerp*, between V_0 and V_1 results in a vector that is no longer normalized and therefore not a valid rotation axis. Of course, this can easily be fixed by re-normalization (*Nlerp*), but then another problem occurs as shown in the left picture of Figure A.3. Since we have, briefly speaking, mapped the equidistant spacing of a straight line onto an arc segment, the distance between two consecutive values of our interpolation parameter is no longer constant over the arc segment.

Instead, we decided to carry out the interpolation as shown on the right side of Figure A.3, thus getting a constant parametrization of the arc segment. If a vector V_p

is created perpendicular to V_0, then V_t and V_1 can be expressed in the following way

$$V_t = \cos(t\alpha)V_0 + \sin(t\alpha)V_p \quad (A.10)$$
$$V_1 = \cos(\alpha)V_0 + \sin(\alpha)V_p \quad (A.11)$$

and it follows from Eq. (A.11) that

$$V_p = \frac{V_1 - \cos(\alpha)V_0}{\sin(\alpha)}.$$

Inserting V_p into Eq. (A.10), then gives

$$\begin{aligned} V_t &= \cos(t\alpha)V_0 + \sin(t\alpha)\cdot\left(\frac{V_1 - \cos(\alpha)V_0}{\sin(\alpha)}\right) \\ &= V_0\frac{(\cos(t\alpha)\cdot\sin(\alpha) - \sin(t\alpha)\cdot\cos(\alpha))}{\sin(\alpha)} + V_1\frac{\sin(t\alpha)}{\sin(\alpha)} \end{aligned}$$

and by applying the addition theorem $\sin(x-y) = \sin x\cdot\cos y - \sin y\cdot\cos x$ to the numerator of the first term, we finally have

$$V_t = \frac{\sin\Big((1-t)\alpha\Big)}{\sin\alpha}V_0 + \frac{\sin(t\alpha)}{\sin\alpha}V_1 \quad (A.12)$$

which is the spherical linear interpolation (*Slerp*) method proposed by Shoemake [Sho85] in the context of quaternion interpolation. We decided to use this formula for the *geometrical* Slerp between two normalized vectors V_0 and V_1.

A.3 Interpolation of Matrix Logarithms

Finally, we have all tools at hand for the interpolation between two deformation gradients J_0 and J_1. The first step is to decompose them into their rotational and scale/ shear part as $J_0 = R_0\cdot S_0$ and $J_1 = R_1\cdot S_1$. In the next step we take the logarithms $log(R_0) = \theta_0 B_0$, $log(R_1) = \theta_1 B_1$, extract their rotation axis from B_0 and B_1 and apply Eq. (A.12). The remaining angles θ_0, θ_1 and the scale/shear parts S_0, S_1 can be interpolated linearly without further treatment. The interpolated gradient J_t thus can be expressed as

$$J_t = e^{(1-t)\log(R_0)+t\log(R_1)}\cdot(1-t)S_0 + tS_1.$$

This scheme is readily suitable for interpolation between more than two input meshes. In case of n compatible meshes the target deformation gradient for each triangle can be calculated as

$$J_t = \exp\left(\sum_{i=1}^{n}\lambda_i\log\left(R_i\right)\right)\cdot\sum_{i=1}^{n}\lambda_i S_i, \quad (A.13)$$

if we require the weights λ_i to sum to unity,

$$\sum_i \lambda_i = 1 \tag{A.14}$$

such that the resulting deformation gradient J_t is described as an affine combination of the input deformation gradients.

If we apply each J_t to the corresponding reference frame (see Fig. A.2), the result will be a triangle soup, since we have interpolated the deformation gradients for each triangle and only gained knowledge how the *spanning vectors* of that triangle were deformed. We do not know the *global* positions of the triangle vertices yet. Hence, one last step is necessary to retrieve the interpolated triangle mesh. As shown e.g. by Sumner et al. [SZGP05] and Botsch et al. [BSPG06] solving a least-squares system for the vertex positions finally yields the desired interpolated mesh.

Bibliography

[AAGU08] Pierre Alliez, Marco Attene, Craig Gotsman, and Giuliana Ucelli. Recent advances in remeshing of surfaces. In L. de Floriani and M. Spagnuolo, editors, *Shape Analysis and Structuring*, pages 53–82. Springer, 2008.

[ABK98] Nina Amenta, Marshall Bern, and Manolis Kamvysselis. A new voronoi-based surface reconstruction algorithm. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH 1998, pages 415–421, 1998.

[ACK01] Nina Amenta, Sunghee Choi, and Ravi Krishna Kolluri. The power crust. In *SMA '01: Proceedings of the sixth ACM symposium on Solid modeling and applications*, pages 249–266, 2001.

[ACOL00] Marc Alexa, Daniel Cohen-Or, and David Levin. As-rigid-as-possible shape interpolation. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00, pages 157–164, 2000.

[AG04] Nizam Anuar and Igor Guskov. Extracting animated meshes with adaptive motion estimation. In B. Girod, M. A. Magnor, and H.-P. Seidel, editors, *Proceedings of Vision, Modeling, and Visualization 2004*, pages 63–71, 2004.

[AG05] Pierre Alliez and Craig Gotsman. Recent advances in compression of 3D meshes. In N. A. Dodgson, M. S. Floater, and M. A. Sabin, editors, *Advances in Multiresolution for Geometric Modelling*, Mathematics and Visualization, pages 3–26. Springer, 2005.

[AH08] Nikolai Alex and Barbara Hammer. Parallelizing single pass patch clustering. In *Proceedings of the European Symposium on Artificial Neural Networks*, pages 227–232, Bruges, April 2008.

[AHB87] K. S. Arun, T. S. Huang, and S. D. Blostein. Least-squares fitting of two 3-D point sets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 9(5):698–700, 1987.

[Ale02a] Marc Alexa. Linear combination of transformations. *ACM Transactions on Graphics*, 21(3):380–387, July 2002. Proceedings of SIGGRAPH.

[Ale02b] Marc Alexa. Recent advances in mesh morphing. *Computer Graphics Forum*, 21(2):173–196, 2002.

[AM00] Marc Alexa and Wolfgang Müller. Representing animations by principal components. *Computer Graphics Forum*, 19(3):411–418, September 2000. Proceedings of Eurographics.

[Ame99] Nina Amenta. The crust algorithm for 3*d* surface reconstruction. In *SCG '99: Proceedings of the fifteenth annual symposium on Computational geometry*, pages 423–424, 1999.

[AS07] R. Amjoun and W. Straßer. Efficient compression of 3D dynamic mesh sequences. *Journal of the WSCG*, 15(1–3):32–46, 2007.

[Bar84] Alan H. Barr. Global and local deformations of solid primitives. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, pages 21–30, 1984.

[BCA07] Yasmine Boulfani-Cuisinaud and Marc Antonini. Motion-based geometry compensation for DWT compression of 3D mesh sequences. In *Proceedings of the IEEE Conference on Image Processing*, volume 1, pages 217–220, San Antonio, TX, September 2007.

[BCAP07] Yasmine Boulfani-Cuisinaud, Marc Antonini, and Frédéric Payan. Motion-based mesh clustering for MCDWT compression of 3D animated meshes. In *Proceedings of the 15th European Signal Processing Conference*, pages 2105–2109, Posnań, Poland, September 2007.

[BM92] Paul J. Besl and Neil D. McKay. A method for registration of 3-D shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, 1992.

[BO05] Jean-Daniel Boissonnat and Steve Oudot. Provably good sampling and meshing of surfaces. *Graphical Models*, 67(5):405–451, 2005.

[BPGK06] Mario Botsch, Mark Pauly, Markus Gross, and Leif Kobbelt. Primo: coupled prisms for intuitive surface modeling. In *Proceedings of Symposium on Geometry Processing 2006*, pages 11–20, Cagliari, Italy, June 2006.

[BSM+03] Hector M. Briceño, Pedro V. Sander, Leonard McMillan, Steven Gortler, and Hugues Hoppe. Geometry videos: a new representation for 3D animations. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 136–146, San Diego, CA, July 2003.

[BSPG06] Mario Botsch, Robert Sumner, Mark Pauly, and Markus Gross. Deformation transfer for detail-preserving surface editing. In L. Kobbelt, T. Kuhlen, T. Aach, and R. Westermann, editors, *Proceedings of Vision, Modeling, and Visualization 2006*, pages 357–364, Aachen, Germany, November 2006. Aka.

[BVGP09] Ilya Baran, Daniel Vlasic, Eitan Grinspun, and Jovan Popović. Semantic deformation transfer. *ACM Transactions on Graphics*, 28(3):Article 36, 6 pages, August 2009. Proceedings of SIGGRAPH.

[BWR+08] Miklós Bergou, Max Wardetzky, Stephen Robinson, Basile Audoly, and Eitan Grinspun. Discrete elastic rods. *ACM Transactions on Graphics*, 27(3):Article 63, 12 pages, August 2008. Proceedings of SIGGRAPH.

[CHHV06] Marie Cottrell, Barbara Hammer, Alexander Hasenfuß, and Thomas Villmann. Batch and median neural gas. *Neural Networks*, 19(6–7):762–771, July–August 2006.

[CL09] Hung-Kuo Chu and Tong-Yee Lee. Multiresolution mean shift clustering algorithm for shape interpolation. *IEEE Transactions on Visualization and Computer Graphics*, 15(5):853–866, 2009.

[CMP+07] G. Charpiat, P. Maurel, J.-P. Pons, R. Keriven, and O. Faugeras. Generalized gradients: Priors on minimization flows. *International Journal of Computer Vision*, 73(3):325–344, 2007.

[CRS98] P. Cignoni, C. Rocchini, and R. Scopigno. Metro: Measuring error on simplified surfaces. *Computer Graphics Forum*, 17(2):167–174, June 1998.

[CS99] S. J. Cunnington and A. J. Stoddart. N-view point set registration: A comparison. In *British Machine Vision Conference*, pages 234–244, 1999.

[DG03] Tamal K. Dey and Samrat Goswami. Tight cocone: a water-tight surface reconstructor. In *SM '03: Proceedings of the eighth ACM symposium on Solid modeling and applications*, pages 127–134, 2003.

[DHKL01] N. Dyn, K. Hormann, S.-J. Kim, and D. Levin. Optimizing 3D triangulations using discrete curvature analysis. In T. Lyche and L. L. Schumaker, editors, *Mathematical Methods for Curves and Surfaces: Oslo 2000*, pages 135–146. 2001.

[DMSB99] Mathieu Desbrun, Mark Meyer, Peter Schröder, and Alan H. Barr. Implicit fairing of irregular meshes using diffusion and curvature flow. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, SIGGRAPH 1999, pages 317–324, 1999.

[DSP06] Kevin G. Der, Robert W. Sumner, and Jovan Popović. Inverse kinematics for reduced deformable models. *ACM Transactions on Graphics*, 25(3):1174–1179, July 2006. Proceedings of SIGGRAPH.

[EDD+95] M. Eck, T. D. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pages 173–182, 1995.

[FDCO03] Shachar Fleishman, Iddo Drori, and Daniel Cohen-Or. Bilateral mesh denoising. *ACM Transactions on Graphics*, 22(3):950–953, 2003.

[FH05] Michael S. Floater and Kai Hormann. Surface parameterization: a tutorial and survey. In Neil A. Dodgson, Michael S. Floater, and Malcolm A. Sabin, editors, *Advances in Multiresolution for Geometric Modelling*, pages 157–186. Springer, 2005.

[Flo98] M. Floater. How to approximate scattered data by least squares. *Technical Report STF42 A98013, SINTEF, Oslo*, 1998.

[Flo03] Michael S. Floater. Mean value coordinates. *Computer Aided Geometric Design*, 20(1):19–27, 2003.

[Gar99] M. Garland. Multiresolution modeling: Survey & future opportunities. In *Proceedings of EUROGRAPHICS*, STAR – State of The Art Reports, pages 111–131, 1999.

[Gee09] Marcus Geelnard. OpenCTM: The open compressed triangle mesh file format. http://openctm.sourceforge.net/, 2009.

[GK04] Igor Guskov and Andrei Khodakovsky. Wavelet compression of parametrically coherent mesh sequences. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 183–192, Grenoble, August 2004.

[GKS00] M. Gopi, S. Krishnan, and C. T. Silva. Surface reconstruction based on lower dimensional localized delaunay triangulation. 2000.

[Gra98] S. Grassia. Practical parameterization of rotations using the exponential map. *Journal of Graphics Tools*, 3(3):29–48, 1998.

[GX02] J. Gallier and D. Xu. Computing exponentials of skew-symmetric matrices and logarithms of orthogonal matrices. *International Journal of Robotics and Automation*, 17(4):173–196, 2002.

[Har75] John A. Hartigan. *Clustering Algorithms*. Wiley Series in Probability and Mathematical Statistics. John Wiley & Sons, New York, 1975.

[HCHY05] Ke-Sen Huang, Chun-Fa Chang, Yu-Yao Hsu, and Shi-Nine Yang. Key probe: a technique for animation keyframe extraction. *The Visual Computer*, 21(8–10):532–541, September 2005.

[HDD+92] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Surface reconstruction from unorganized points. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '92, pages 71–78, 1992.

[HDD+93] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, pages 19–26, 1993.

[HHS08] P. Huang, A. Hilton, and J. Starck. Automatic 3D video summarization: Key frame extraction from self-similarity. In *Proceedings of the Forth International Symposium on 3D Data Processing, Visualization and Transmission*, Atlanta, GA, June 2008.

[Hig86] Nicholas J Higham. Computing the polar decomposition with applications. *SIAM Journal on Scientific and Statistical Computing*, 7(4):1160–1174, October 1986.

[HSS+09] N. Hasler, C. Stoll, M. Sunkel, B. Rosenhahn, and H.-P. Seidel. A statistical model of human pose and body shape. *Computer Graphics Forum*, 2(28), March 2009. Proceedings of Eurographics 2009.

[Hur09] Philippe Hurbain. A nxt-based laser sweeping mechanism used to digitize 3d shapes. http://www.philohome.com/scan3d/scan3d.htm, 2009.

[IAS02] Martin Isenburg, Pierre Alliez, and Jack Snoeyink. A benchmark coder for polygonal mesh compression. http://www.cs.unc.edu/~isenburg/pmc/, 2002.

[IR03] Lawrence Ibarria and Jarek Rossignac. Dynapack: space-time compression of the 3D animations of triangle meshes with fixed connectivity. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 126–135, San Diego, CA, July 2003.

[JDD03] Thouis R. Jones, Frédo Durand, and Mathieu Desbrun. Non-iterative, feature-preserving mesh smoothing. *ACM Transactions on Graphics*, 22(3):943–949, 2003. Proceedings of SIGGRAPH.

[KBH06] Michael Kazhdan, Matthew Bolitho, and Hugues Hoppe. Poisson surface reconstruction. In *SGP '06: Proceedings of the fourth Eurographics Symposium on Geometry Processing*, pages 61–70, 2006.

[KG04] Zachi Karni and Craig Gotsman. Compression of soft-body animation sequences. *Computers & Graphics*, 28(1):25–34, February 2004.

[KG08] Scott Kircher and Michael Garland. Free-form motion processing. *ACM Transactions on Graphics*, 27(2):1–13, 2008.

[KKS95] Myoung-Jun Kim, Myung-Soo Kim, and Sung Yong Shin. A general construction scheme for unit quaternion curves with simple high order derivatives. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pages 369–376, New York, NY, USA, 1995. ACM.

[KMP07] Martin Kilian, Niloy J. Mitra, and Helmut Pottmann. Geometric modeling in shape space. *ACM Transactions on Graphics*, 26(3), July 2007. Proceedings of SIGGRAPH.

[KS04] Vladislav Kraevoy and Alla Sheffer. Cross-parameterization and compatible remeshing of 3D models. *ACM Transactions on Graphics*, 23(3):861–869, 2004. Proceedings of SIGGRAPH.

[Law61] C. L. Lawson. *Contributions to the Theory of Linear Least Maximum Approximation*. PhD thesis, University of California, Los Angeles, CA, 1961.

[LC87] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '87, pages 163–169, 1987.

[Lee08] Jehee Lee. Representing rotations and orientations in geometric computing. *Computer Graphics and Applications*, 28(2):75–83, March/April 2008.

[Len99] Jerome Edward Lengyel. Compression of time-dependent geometry. In *Proceedings of the 1999 Symposium on Interactive 3D Graphics*, pages 89–95, Atlanta, GA, April 1999.

[LKT+07] Pai-Feng Lee, Chi-Kang Kao, Juin-Ling Tseng, Bin-Shyan Jong, and Tsong-Wuu Lin. 3D animation compression using affine transformation matrix and principal component analysis. *IEICE Transactions on Information and Systems*, E90-D(7):1073–1084, July 2007.

[LLWC08] Tong-Yee Lee, Chao-Hung Lin, Yu-Shuen Wang, and Tai-Guang Chen. Animation key-frame extraction and simplification using deformation analysis. *IEEE Transactions on Circuits and Systems for Video Technology*, 18(4):478–486, April 2008.

[LSLCO05] Yaron Lipman, Olga Sorkine, David Levin, and Daniel Cohen-Or. Linear rotation-invariant coordinates for meshes. *ACM Transactions on Graphics*, 24(3):479–487, 2005. Proceedings of SIGGRAPH.

[LT01] Ik Soo Lim and Daniel Thalmann. Key-posture extraction out of human motion data. In *Proceedings of the 23rd Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, volume 2, pages 1167–1169, Istanbul, October 2001.

[LTJW07] Ligang Liu, Chiew-Lan Tai, Zhongping Ji, and Guojin Wang. Non-iterative approach for global mesh optimization. *Computer-Aided Design*, 39(9):772–782, 2007.

[MA97] David Mount and Sunil Arya. Ann: A library for approximate nearest neighbor searching. http://www.cs.umd.edu/~mount/ANN/, 1997.

[MBS93] T. M. Martinetz, S. G. Berkovich, and K. J. Schulten. "Neural-gas" network for vector quantization and its application to time-series prediction. *IEEE Transactions on Neural Networks*, 4(4):558–569, July 1993.

[MdGD+10] Patrick Mullen, Fernando de Goes, Mathieu Desbrun, David Cohen-Steiner, and Pierre Alliez. Signing the unsigned: Robust surface reconstruction from raw pointsets. *Computer Graphics Forum*, 29(5):1733–1741, July 2010.

[MHTG05] Matthias Müller, Bruno Heidelberger, Matthias Teschner, and Markus Gross. Meshless deformations based on shape matching. *ACM Transactions on Graphics*, 24(3):471–478, July 2005. Proceedings of SIGGRAPH.

[ML78] Cleve Moler and Charles Van Loan. Nineteen dubious ways to compute the exponential of a matrix. *SIAM Review*, 20:801–836, 1978.

[ML03] Cleve Moler and Charles Van Loan. Nineteen dubious ways to compute the exponential of a matrix, twenty-five years later. *SIAM Review*, 45:3–49, 2003.

[MLBD02] M. Meyer, H. Lee, A. H. Barr, and M. Desbrun. Generalized barycentric coordinates for irregular polygons. *Journal of Graphics Tools*, 7(1):13–22, 2002.

[MS94] Thomas Martinetz and Klaus Schulten. Topology representing networks. *Neural Networks*, 7(3):507–522, 1994.

[MW59] T. S. Motzkin and J. L. Walsh. Polynomials of best approximation on a real finite point set. *Transactions of the American Mathematical Society*, 91:231–245, 1959.

[MW09] Sven Molkenstruck and Simon Winkelbach. David-laserscanner. http://www.david-laserscanner.com/, 2009.

[NISA06] Andrew Nealen, Takeo Igarashi, Olga Sorkine, and Marc Alexa. Laplacian mesh optimization. In *Proceedings of GRAPHITE '06*, pages 381–389, 2006.

[OB01] Yutaka Ohtake and Alexander Belyaev. Mesh optimization for polygonized isosurfaces. *Computer Graphics Forum*, 20(3):368–376, 2001.

[OWT02] Steven J. Owen, David R. White, and Timothy J. Tautges. Facet-based surfaces for 3D mesh generation. In *Proceedings of the 11th International Meshing Roundtable*, pages 297–312, September 2002.

[PA05] F. Payan and M. Antonini. Wavelet-based compression of 3D mesh sequences. In *Proceedings of the Second International Conference on Machine Intelligence*, Tozeur, Tunisia, November 2005.

[Pea01] Karl Pearson. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 2:559–572, 1901.

[PKK05] Jingliang Peng, Chang-Su Kim, and C. C. Jay Kuo. Technologies for 3D mesh compression: A survey. *Journal of Visual Communication and Image Representation*, 16(6):688–733, December 2005.

[PLH02] H. Pottmann, S. Leopoldseder, and M. Hofer. Simultaneous registration of multiple views of a 3D object. In Rainer Kalliany and Franz Leberl, editors, *Photogrammetric Computer Vision*, volume 34, Part 3A of *Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, pages 265–270. ISPRS, 2002.

[PP93] Ulrich Pinkall and Konrad Polthier. Computing discrete minimal surfaces and their conjugates. *Experimental Mathematics*, 2(1):15–36, 1993.

[PS04] Min Je Park and Sung Yong Shin. Example-based motion cloning. *Journal of Visualization and Computer Animation*, 15(3–4):245–257, July 2004.

[Ros04] Jarek Rossignac. Surface simplification and 3D geometry compression. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 54, pages 1209–1240. CRC Press, Boca Raton, FL, second edition, 2004.

[Row98] Sam Roweis. EM algorithms for PCA and SPCA. In *Proceedings of the 1997 conference on Advances in Neural Information Processing Systems 10*, NIPS '97, pages 626–632, Cambridge, MA, USA, 1998. MIT Press.

[RY07] Laurent Rineau and Mariette Yvinec. A generic software design for delaunay refinement meshing. *Comput. Geom. Theory Appl.*, 38(1-2):100–110, 2007.

[SA07] O. Sorkine and M. Alexa. As-rigid-as-possible surface modeling. In *Proceedings of SGP 2007*, pages 109–116, 2007.

[SAG03] Vitaly Surazhsky, Pierre Alliez, and Craig Gotsman. Isotropic remeshing of surfaces: a local parameterization approach. In *Proceedings of the 11th International Meshing Roundtable*, pages 215–224, September 2003.

[SAG+05] Peter Shirley, Michael Ashikhmin, Michael Gleicher, Stephen Marschner, Erik Reinhard, Kelvin Sung, William Thompson, and Peter Willemsen. *Fundamentals of Computer Graphics, Second Ed.* A. K. Peters, Natick, MA, USA, 2005.

[SAPH04] John Schreiner, Arul Asirvatham, Emil Praun, and Hugues Hoppe. Inter-surface mapping. *ACM Transactions on Graphics*, 23(3):870–877, 2004. Proceedings of SIGGRAPH.

[SD92] Ken Shoemake and Tom Duff. Matrix animation and polar decomposition. In *Proceedings of Graphics Interface '92*, pages 258–264, Vancouver, BC, May 1992.

[SG03] Vitaly Surazhsky and Craig Gotsman. Explicit surface remeshing. In *Proceedings of SGP 2003*, pages 20–30, 2003.

[SGWM93] Thomas W. Sederberg, Peisheng Gao, Guojin Wang, and Hong Mu. 2-d shape blending: an intrinsic solution to the vertex path problem. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, pages 15–18, 1993.

[SH96] A. J. Stoddart and A. Hilton. Registration of multiple point sets. In *Proc. 13 th Int. Conf. on Pattern Recognition*, pages 40–44, 1996.

[Sho85] Ken Shoemake. Animating rotation with quaternion curves. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '85, pages 245–254, 1985.

[SK04] Alla Sheffer and Vladislav Kraevoy. Pyramid coordinates for morphing and deformation. In *3DPVT '04: Proceedings of the 3D Data Processing, Visualization, and Transmission, 2nd International Symposium*, pages 68–75, Washington, DC, USA, 2004. IEEE Computer Society.

[SP04] Robert W. Sumner and Jovan Popović. Deformation transfer for triangle meshes. *ACM Transactions on Graphics*, 23(3):399–405, 2004. Proceedings of SIGGRAPH.

[SPK+07] R. C. Smith, R. Pawlicki, I. Kókai, J. Finger, and T. Vetter. Navigating in a shape space of registered models. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1552–1559, 2007.

[SSK05] Mirko Sattler, Ralf Sarlette, and Reinhard Klein. Simple and efficient compression of animation sequences. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 209–217, Los Angeles, CA, July 2005.

[Sum06] Robert W. Sumner. *Mesh modification using deformation gradients*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2006.

[SZGP05] Robert W. Sumner, Matthias Zwicker, Craig Gotsman, and Jovan Popović. Mesh-based inverse kinematics. *ACM Transactions on Graphics*, 24(3):488–495, July 2005. Proceedings of SIGGRAPH.

[Tau95] G. Taubin. A signal approach to fair surface design. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pages 351–358, 1995.

[Tol03] Sivan Toledo. Taucs: A library of sparse linear solvers, version 2.2. http://www.tau.ac.il/~stoledo/taucs/, 2003.

[vDA95] R. van Damme and L. Alboul. Tight triangulations. In M. Dæhlen, T. Lyche, and L. L. Schumaker, editors, *Mathematical Methods for Curves and Surfaces*, pages 517–526. 1995.

[Wac75] E. L. Wachspress. *A Rational Finite Element Basis*. Academic Press, New York, 1975.

[WB00] J. A. Williams and M. Bennamoun. Simultaneous registration of multiple point sets using orthonormal matrices. In *ICASSP '00: Proceedings of the Acoustics, Speech, and Signal Processing, 2000. on IEEE International Conference*, pages 2199–2202, Washington, DC, USA, 2000. IEEE Computer Society.

[WDAH10] Tim Winkler, Jens Drieseberg, Marc Alexa, and Kai Hormann. Multi-scale geometry interpolation. *Computer Graphics Forum*, 29(2):309–318, May 2010. Proceedings of Eurographics.

[WDH+08] Tim Winkler, Jens Drieseberg, Alexander Hasenfuß, Barbara Hammer, and Kai Hormann. Thinning mesh animations. In O. Deussen, D. Keim, and D. Saupe, editors, *Proceedings of Vision, Modeling, and Visualization 2008*, pages 149–158, Konstanz, Germany, October 2008. Aka.

[WHG08] Tim Winkler, Kai Hormann, and Craig Gotsman. Mesh massage: A versatile mesh optimization framework. *The Visual Computer*, 24(7–9):775–785, July 2008. Best Papers of CGI 2008.

[XZWB05] Dong Xu, Hongxin Zhang, Qing Wang, and Hujun Bao. Poisson shape interpolation. In *Proceedings of the 2005 ACM Symposium on Solid and Physical Modeling*, pages 267–274, Cambridge, MA, June 2005.

[YKL02] Jeong-Hyu Yang, Chang-Su Kim, and Sang Uk Lee. Compression of 3-D triangle mesh sequences based on vertex-wise motion vector prediction. *IEEE Transactions on Circuits and Systems for Video Technology*, 12(12):1178–1184, December 2002.